VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

PETRIHO SÍTĚ V NÁSTROJI NETLAB

PETRI NETS IN NETLAB TOOL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

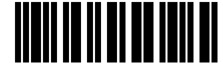
AUTOR PRÁCE | Bc. ONDREJ ŠAJDÍK
AUTHOR

VEDOUCÍ PRÁCE | doc. Mgr. ADAM ROGALEWICZ, Ph.D.
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce



144951

Ústav: Ústav inteligentních systémů (UITS)
Student: **Šajdík Ondrej, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Strojové učení
Název: **Petriho sítě v nástroji Netlab**
Kategorie: Algoritmy a datové struktury
Akademický rok: 2022/23

Zadání:

1. Seznamte se s nástrojem Netlab, s jeho funkčností a zdrojovými kódy. Prostudujte základní algoritmy pro analýzu P/T Petriho sítí. Zaměřte se na techniku zpětné analýzi v Petriho sítích, strom pokrytí a výpočet invariantů.
2. Proveďte komplexní restrukturalizaci a refaktoring původního nástroje Netlab do standardu C++11 (nebo vyššího). V rámci restrukturalizace transformujte části kódu tak, aby byly spustitelné pod běžným uživatelským účtem. Provádějte restrukturalizaci s ohledem na další snadnou rozšiřitelnost.
3. Doplňte implementaci algoritmu zpětné analýzy Petriho sítě.
4. Diskutujte možnosti dalšího rozšítření. V technické zprávě zdokumentujte postup při rozšiřování o další moduly.

Literatura:

- Češka, M.: Petriho sítě, Akad.nakl. CERM, Brno, 1994. ISBN: 8-085-86735-4

Při obhajobě semestrální části projektu je požadováno:
Bod 1 a část bodu 2.

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

Vedoucí práce: **Rogalewicz Adam, doc. Mgr., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 17.5.2023
Datum schválení: 3.11.2022

Abstrakt

Cieľom tejto práce je podporiť pokračovanie vývoja nástroja Netlab a zjednodušiť jeho rozšírenie. Netlab je desktopová aplikácia, ktorá umožňuje modelovanie systémov vo forme Petriho sieti prostredníctvom grafického editora a následovne vykonávať model checking analýzy na vytvorenom modeli. V rámci práce boli preskúmané koncepty model checkingu, Petriho sieti a dostupných analýz v aplikácii. Aplikácia nebola dlhodobo vyvíjaná, čo spôsobilo značné prekážky pre ďalšie vylepšenia. V rámci tejto práce sú tieto problémy adresované a riešené. Ďalším cieľom je umožniť užívateľom aplikácie spustiť nad vytvoreným modelom algoritmus spätnej analýzy, ktorý bol implementovaný, čím sa demonštruje rozšíriteľnosť a možnosti ďalšieho vývoja.

Abstract

The aim of this work is to support the continuation of the Netlab tool's development and to simplify its extension. Netlab is a desktop application that allows for the modeling of systems in the form of Petri nets using a graphical editor, and subsequently performs model checking analyses on the created model. Within the scope of this work, the concepts of model checking, Petri nets, and available analyses within the application were examined. The application had not been developed for an extended period, resulting in significant obstacles for further improvements. These issues are addressed and resolved throughout the course of this work. Another goal is to enable users of the application to execute a reverse analysis algorithm on the created model, which has been implemented, thereby demonstrating extensibility and potential for future development.

Kľúčové slová

Petriho siete, Netlab, spätná analýza, testovanie, verifikácia, modelovanie, pokrytelnosť, dosiahnuteľnosť

Keywords

Petri nets, Netlab, backward analysis, testing, verification, modeling, coverability, reachability

Citácia

ŠAJDÍK, Ondrej. *Petriho sítě v nástroji Netlab*. Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Mgr. Adam Rogalewicz, Ph.D.

Petriho sítě v nástroji Netlab

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením doc. Mgr. Adam Rogalewicz, Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

....
Ondrej Šajdík
17. mája 2023

Poďakovanie

Moje poďakovanie patrí vedúcemu práce, Mgr. Adam Rogalewicz, Ph.D, za odborné vedenie tejto práce, cenné rady a ochotu poskytnutú pri konzultáciách.

Obsah

1 Úvod **5**

2 Model checking **6**
- 2.1 Formálna verifikácia systému . . . 6
- 2.2 Model checking . . . 7
  - 2.2.1 Prístup . . . 7
  - 2.2.2 Hrubá sila . . . 7
- 2.3 Model checking proces . . . 7
  - 2.3.1 Modelovanie . . . 8
  - 2.3.2 Beh algoritmu . . . 9
  - 2.3.3 Analýza . . . 9

3 Petriho Siete **10**
- 3.1 Základný koncept . . . 10
  - 3.1.1 Štruktúra siete . . . 10
  - 3.1.2 Značenie . . . 11
  - 3.1.3 Vykonanie prechodu . . . 12
  - 3.1.4 Kapacita . . . 13
- 3.2 P/T Petriho siete . . . 14
- 3.3 Rozšírenia Petriho sietí . . . 14
  - 3.3.1 Inhibítory . . . 14
  - 3.3.2 Priorita . . . 15
  - 3.3.3 Čas . . . 15
  - 3.3.4 Stochastické petriho siete . . . 16
  - 3.3.5 Farebné Petriho siete . . . 16

4 Analýzy Petriho sieti **17**
- 4.1 Vlastnosti . . . 17
  - 4.1.1 Bezpečnosť (Safeness) . . . 17
  - 4.1.2 Obmedzenosť (Boundness) . . . 17
  - 4.1.3 Živosť (Liveness) . . . 17
  - 4.1.4 Konzervatívnosť (Conservation) . . . 18
- 4.2 Problémy . . . 18
  - 4.2.1 Problém dosiahnuteľnosti . . . 19
  - 4.2.2 Problém pokryteľnosti . . . 19
- 4.3 Techniky pre analýzu . . . 19
  - 4.3.1 Maticové analýzy . . . 19
  - 4.3.2 Strom dosiahnuteľných značení . . . 21

4.3.3 Spätná analýza pokryteľnosti 24

5 MFC framework **25**
5.1 Document/View architektúra 25
5.1.1 Aplikácia 26
5.1.2 Okno 26
5.1.3 Dokument 26
5.1.4 Zobrazenie 26
5.2 Správy 26
5.2.1 Mapovanie správ 27
5.3 Kreslenie 28
5.3.1 Device context 28
5.3.2 GDI Objects 29

6 Netlab **30**
6.1 Nástroj Netlab 30
6.2 Grafický editor 31
6.3 Simulácia 31
6.4 Invarianty a grafová analýza 31
6.4.1 Invarianty 31
6.4.2 Grafy 32
6.4.3 Výsledky analýzy 32
6.5 Import/Export 32

7 Netlab implementácia a vývojové prostredie **33**
7.1 Vývojové prostredie 33
7.1.1 Pôvodné vývojové prostredie 33
7.1.2 Prechod na nové prostredie 33
7.1.3 Matlab a Perl 34
7.1.4 Projekty 35
7.2 Kvalita kódu 35
7.3 Architektúra 36
7.3.1 Model 36
7.3.2 Analýza 38
7.3.3 MFC aplikácia 39

8 Implementácia spätnej analýzy **41**
8.1 Spustenie 41
8.2 Beh 42
8.2.1 Reprezentácia stavu 42
8.2.2 Algoritmus 42
8.2.3 Generovanie stavov 43
8.3 Reprezentácia výsledkov 44
8.4 Testovanie 44

9 Záver **45**

Literatúra **46**

A Príprava prostredia a kompilácia **48**
A.1 Prerekvizity 48
A.2 Kompilácia a spustenie 48
A.3 Testy 48

Zoznam obrázkov

2.1 Model checking . . . 7
2.2 Model checking proces . . . 8

3.1 Príklad Petriho siete . . . 11
3.2 Príklady značenia miest . . . 11
3.3 Príklad vykonateľných a nevykonateľných prechodov . . . 12
3.4 Príklad vykonania prechodu . . . 13
3.5 Generátor značiek . . . 13
3.6 Príklad jednoduchej siete s kapacitou. . . . 13
3.7 Príklad prevodu siete s kapacitou na sieť s komplementárnym miestom. . . 14
3.8 Príklad použitia inhibítoru. . . . 15

4.1 Príklad siete s prechodmi rôznej úrovne živosti. . . . 18
4.2 Príklad konzervatívnej siete. . . . 20
4.3 Prvý krok tvorby stromu dosiahnuteľných značení . . . 22
4.4 Druhý krok tvorby stromu dosiahnuteľných značení . . . 22
4.5 Strom dosiahnuteľných značení siete 4.2 . . . 22
4.6 Príklad Petriho siete, ktorá nie je obmedzená . . . 23
4.7 Strom dosiahnuteľných značení siete 4.6 . . . 23

6.1 Nástroj Netlab . . . 30
6.2 Dialógové okná pre úpravu vlastností siete. . . . 31
6.3 Textová reprezentácia grafu pokryteľnosti. . . . 32

7.1 Časti Netlab aplikácie a ich závislosti. . . . 36
7.2 Závislosti medzi triedami reprezentujúce prvky siete. . . . 37
7.3 Závislosti kontajnerov pre ukladanie zoznamov prvkov Petriho siete. . . . 37
7.4 Časti grafického rozhrania Netlabu . . . 39

Kapitola 1

Úvod

Model checking je technika používaná na overenie správnosti systému alebo softvérového programu. Jedná sa o silný nástroj pre zvýšenie istoty, že tieto systémy budú fungovať v súlade s požiadavkami, bez nechceného správania alebo chýb. Predstavuje spôsob ako zabezpečiť správnosť a spoľahlivosť systémov a softvérových programov, najmä v prípadoch, kedy sú tieto systémy kritické z hľadiska bezpečnosti alebo hospodárenia s finančnými prostriedkami. Táto technika funguje tak, že sa vytvára model systému vo forme, ktorá sa dá ďalej analyzovať pomocou špecializovaných softvérových nástrojov.

Jedným z týchto nástrojov je Netlab. Desktopová aplikácia poskytujúca užívateľské rozhranie umožňujúce tvorbu a analýzu modelov vo forme Petriho sieti, matematického nástroja založeného na koncepte smerovaných grafov pre modelovanie a analýzu distribuovaných systémov, procesov a operácii. Aplikácia umožňuje užívateľovi modelovanie cez grafický editor a na takto vytvorenom modeli spúšťať analýzy, zahŕňajúce graf dosiahnuteľnosti alebo detekcia invariantov.

Táto práca sa zaoberá práve týmto nástrojom. Nachádza sa tu vysvetlenie základných princípov model checkingu, Petriho sieti a nad nimi implementovaných analýz. Pretože nástroj nebol dlhodobo vyvíjaný, tak sa táto práca ďalej zaoberá umožneniu ďalšieho vývoja v podobe prechodu do vývojového prostredia používajúce súčasne podporované prostriedky ako aj reštrukturalizáciu projektu a vysvetlenie jednotlivých častí. Práca rieši rozšírenie aplikácie o implementáciu spätnej analýzy, pre overenie pokryteľnosti špecifického stavu. Na tejto implementácii je okrem toho ukázaný spôsob pre pridanie analýzy, ich testovania a prvkov užívateľského rozhrania.

Kapitola 2

Model checking

V tejto kapitole bude uvedená problematika model checkingu, ako jednej z metód pre formálnu verifikáciu systému.

2.1 Formálna verifikácia systému

Jedným z najpoužívanejších prístupov pre overenie správnosti vlastností systému je testovanie. To pozostáva zo spúšťania systému s určitými vstupmi a následovné porovnanie výsledkov s očakávanými. Podmienkou je mať hotovú aspoň čiastočne spustiteľnú verziu systému, čo pri určitých typoch projektov, obzvlášť u hardwardových, môže byť až v neskoršej fáze vývojového cyklu. Zároveň aj keď testovanie dokáže nájsť chyby v systéme, tak stále nedokáže zaručiť, že tam žiadne nie sú. Alternatívou k tomu je práve formálna verifikácia.

Formálna verifikácia je proces dokazovania správnosti algoritmu alebo obecne nejakého systému z definovanej špecifikácie za pomoci aplikácie formálnych metód. Za správnosť sa môže považovať napríklad bezpečnosť, spoľahlivosť alebo dostupnosť. [11] Metód pre formálnu verifikáciu existuje viacero. Medzi hlavné dve však patria následujúce:

- **Dokazovanie teorémou** (Theorem Proving). Vlastnosti a chovanie systému spolu s prostredím, i ďalšími znalosťami sú reprezentované sadou logických formúl. Zisťuje sa či logické formule sú voči sebe konzistentné alebo existuje spor. [9]

- **Model Checking**. Metóda prehľadávania všetkých možných stavov konečného modelu pre danú vlastnosť. Narozdiel od dokazovania teorémou sú chovanie a vlastnosti systému popísané modelom. Používa sa napríklad v dizajne digitálnych obvodov a softvérových systémov.

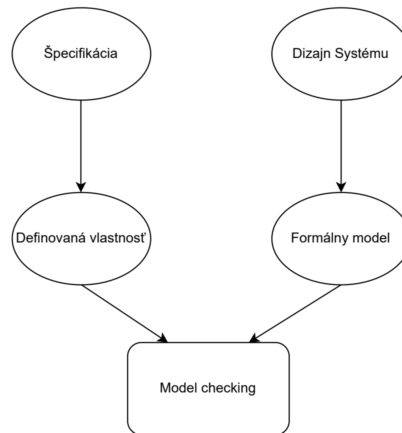
Na rozdiel od testovania sa formálna verifikácia snaží dokázať danú vlastnosť a ak skončí sporom, tak za predpokladu, že verifikácia bola vykonaná správne to znamená prítomnosť chyby v uvažovanom systéme. Samotný spor obsahuje protipríklad, ktorý potom ukazuje ako danú chybu zreprodukovať. Platí, že nie je potrebný spustiteľný systém, takže je možné ju vykonať už v skorších fázach vývoja. To často vyžaduje extra úsilie a výpočtové zdroje zdražujúc vývoj. Na druhú stranu formálna verifikácia nedokáže zaistiť správnosť implementácie, či výroby, a preto sa v praxi môžu používať oba prístupy spolu.

2.2 Model checking

Technika overovania vlastností systému pomocou prehľadávania všetkých možných stavov systému sa nazýva model checking.[2]

2.2.1 Prístup

Užívateľ vytvorí formálny model systému a špecifikuje vlastnosť, ktorú od systému požaduje. Tento model a definovanú vlastnosť vloží na vstup model checking algoritmu. Ten pomocou hrubej sily systematicky prehľadáva stavový priestor a tak sa snaží dokázať, či systém splňuje definovanú vlastnosť. Viz. 2.1.



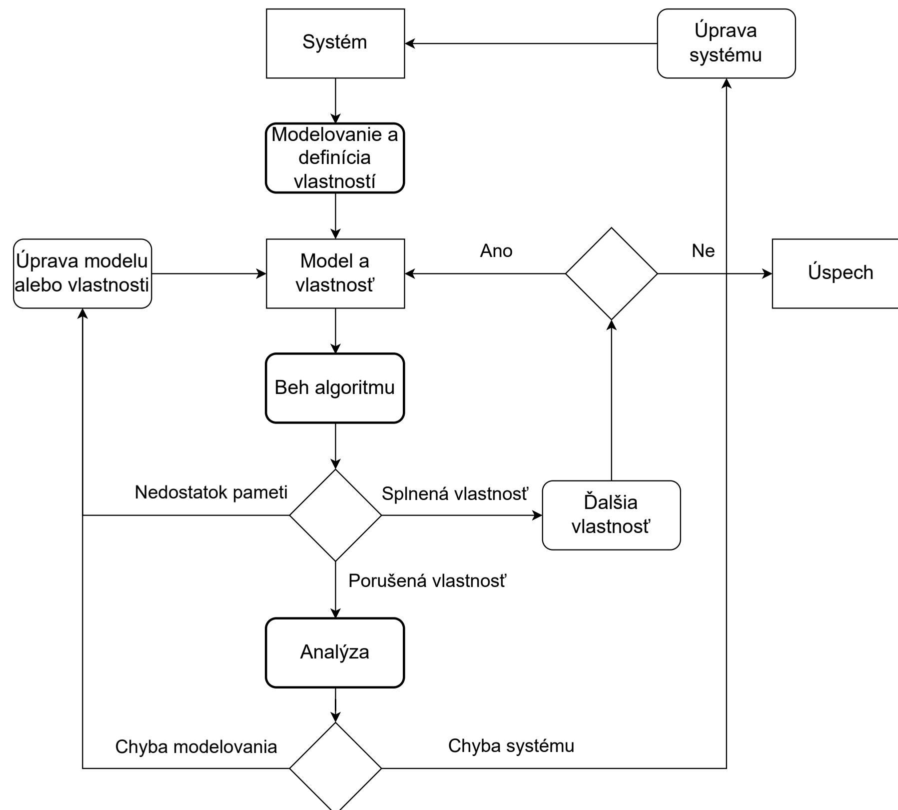
Obr. 2.1: Model checking

2.2.2 Hrubá sila

Často sa používajú algoritmy za použitia hrubej sily, s nie práve nízkou časovou i priestorovou zložitosťou. Hlavným problémom sú u komplexnejších systémov práve výpočtové zdroje. Teda existuje reálne obmedzenie na komplexnosť verifikovaného systému. Za pomoci algoritmov s optimalizovanými dátovými štruktúrami a efektívnou technikov prehľadávania stavového priestoru je síce možné hranicu tohto obmedzenia posunúť, ale kvalita vytvoreného modelu hraje v tomto tiež veľkú rolu.

2.3 Model checking proces

Proces je znázornený na obrázku 2.2 a skladá sa z troch hlavných častí. Modelovanie, spúšťanie a analýza.



Obr. 2.2: Model checking proces

2.3.1 Modelovanie

Najprácnejšia a najdôležitejšia časť model checking procesu. Modelovanie je fáza, v ktorej sa pripravujú vstupy pre verifikačný algoritmus. Vstupy sú ako už bolo popísané vyššie nutné vytvoriť práve dva. Model a overovaná vlastnosť. Platí, že výsledky môžu byť len tak kvalitné, ako sú vstupy.

Model zapísaný vo formálnom modelovacom jazyku akceptovaným algoritmom. Správny model presne a jednoznačne popisuje chovanie uvažovaného systému. Nemusí ho popisovať 1:1, a často sa pri prevode na modelovací jazyk môže zjednodušiť, prípadne skomplikovať a tak verifikovanie či model odpovedá uvažovanému systému je problém sám o sebe. Väčšinou sa jedná o popis pozostávajúci z konečného počtu stavov a prechodov medzi nimi. Nie je to však pravidlo. Po jeho vytvorení je vhodné otestovať správnosť nejakou formou simulácie aby sa odstránili chyby modelu, ktoré by mohli skomplikovať beh verifikačného algoritmu a viedli na chybné výsledky.

Vlastnosť popísaná vo vhodnej logike akceptovanej verifikačným algoritmom. Napríklad pomocou logickej formule. Vychádza z požiadavkov na systém a je možné overovať široké množstvo vlastností.[2] Patria sem napríklad nasledovné:

- Správna funkčnosť. Systém robí to, čo má.
- Dosiahnuteľnosť. Je možné dosiahnuť špecifického stavu. Napr. uviaznutie systému.
- Bezpečnosť. Nie je možné sa dostať do zlého stavu.
- Spoľahlivosť. Vždy sa nakoniec môže dostať do úspešného stavu.

- Férovosť. Daný stav je dosiahnutý opakovane pri špecifických podmienkach.
- Vlastnosti v reálnom čase. Napríklad, odpoveď na správu je vygenerovaná do desiatich sekúnd.

2.3.2 Beh algoritmu

Ako už bolo spomínané, tak sa často jedná o brute-force algoritmy, takže táto časť môže zabrať veľa času a zdrojov. Inak je táto fáza celkom priamočiara. Algoritmus sa inicializuje s pripravenými vstupmi a pomocou prehľadávania stavového priestoru dospeje k výsledkom.

2.3.3 Analýza

Koniec behu algoritmu vedie na práve jeden z troch výsledkov:

1. **Úspech.** Vlastnosť je splnená. Môže sa prejsť k overovaniu ďalšej vlastnosti.
2. **Neúspech.** Vlastnosť nie je splnená. Výsledkom je protipríklad. V takom prípade je potrebné protipríklad zanalyzovať simuláciou. Chyba mohla byť spôsobená chybným modelom alebo v zadanej vlastnosti. Ak tomu tak nie je, tak sa odhalila chyba dizajnu. Ak chyba nebola spôsobená zle zadanou vlastnosťou, tak je po úprave potrebné spustiť znova analýzu všetkých vlastností znova, pretože zmena by na ne mohla mať vplyv.
3. **Nedostatok pamäti.** Model je príliš veľký a preto bol beh algoritmu prerušený z dôvodu nedostatku pamäti. Toto je možné riešiť rôznymi spôsobmi.
    - Optimalizovať pamäťové nároky pomocou symbolických dátových štruktúr.
    - Zvýšiť úroveň abstrakcie modelu k uvažovanému systému
    - Vytvoriť špecifické modely pre konkrétne vlastnosti
    - Ak je prijateľná nižšia presnosť výsledkov, potom riešením môže byť prehľadanie iba časti stavového priestoru.

Kapitola 3

Petriho Siete

V tejto kapitole je popísaný koncept jedného z modelovacích jazykov používaný pre model checking. Obsahuje formálnu definíciu Petriho sieti i popis grafickej reprezentácie spolu s príkladmi.

3.1 Základný koncept

Teória k Petriho sieťam má pôvod v dizertačnej práci nemeckého vedca Carla Adama Petriho. [16] Formálny jazyk Petriho sieti bol odvtedy vyvíjaný a používaný na teoretickej, ale i praktickej úrovni. Základný koncept sa však nedal uplatniť všade a preto pre zvýšenie vyjadrovacej sily, bolo vyvinutých viacero rozšírení. [4] Niektoré z týchto rozšírení sú uvedené v podkapitole 3.3.

3.1.1 Štruktúra siete

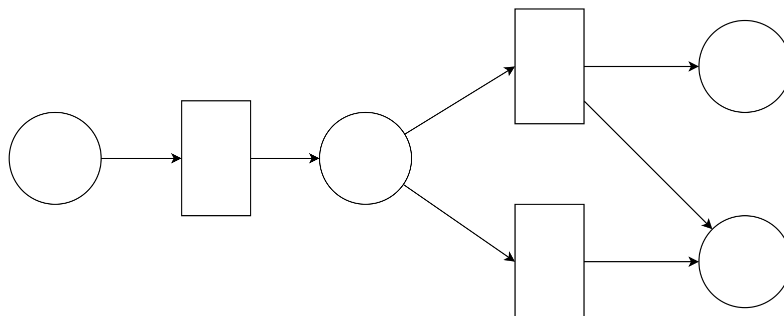
Petriho siete sú matematický modelovací jazyk a grafický nástroj. [14] Jedná sa o orientovaný graf s dvomi typmi uzlov. Petriho sieť teda pozostáva z troch následujúcich komponent:

1. **Miesto**. Graficky reprezentované ako kruhy alebo elipsy sú vždy pasívne komponenty. Miesta vyjadrujú stav systému.

2. **Prechod**. Graficky reprezentované ako obdĺžniky a jedná sa o aktívne komponenty. Menia stav a predstavujú udalosť v systéme.

3. **Hrany** spájajú miesta s prechodmi a naopak. Nikdy sa nejedná o prepojenie medzi dvomi miestami, alebo dvomi prechodmi.

[19] Príklad takejto siete je možné vidieť na obrázku 3.1.



Obr. 3.1: Príklad Petriho siete

Formálne je Petriho sieť definovaná ako trojica $N = (P, T, F)$, kde:

- P je konečná množina miest. (Places)
- T je je konečná množina prechodov. (Transitions)
- $F \subseteq (P \times T) \cup (T \times P)$ je nazývaná toková relácia. (Flow relation)

a platí, že:

- $P \cap T = \emptyset$
- F je binárna relácia.

Pre všetky prvky $x \in (P \cup T)$ existuje vstupná a výstupná množina prvkov.

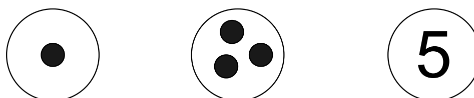
- Vstupná množina ${}^{\bullet}x$ je množina prvkov, kde hrana smeruje do x. ${}^{\bullet}x = \{y|yFx\}$
- Výstupná množina $x^{\bullet}$ je množina prvkov, kde hrana smeruje od x. $x^{\bullet} = \{y|xFy\}$

Vstupná množina poskytuje informácie o prvkoch, ktoré môžu povoliť prvok x. U prechodu to znamená miesta, ktoré musia mať značenie na odobranie a u miesta to značí prechody, ktoré mu môžu značenie poskytnúť.

Výstupná množina poskytuje informácie o prvkoch, ktoré sú povolené prvkom x. U prechodu sa jedná o miesta kam sa pridá značenie. Pre miesto je to prechod, ktorý mu značenie odoberie.

3.1.2 Značenie

Okrem siete samotnej Petriho siete obsahujú aj indikátor stavu, ktorý nazývame značenie. To v závislosti od rozšírenia, ktoré používame môže byť rôzne komplexné. Tu sa bude zaoberať tým najprimitívnejším typom a to graficky znázornený ako čierna značka, ktorá buď v mieste je alebo nie je. Značenie celkovo teda predstavuje rozloženie značiek medzi miestami. [19] V jednom mieste sa však môže nachádzať viacero značiek naraz. Aj preto sa v takých prípadoch používajú jednoducho číslice vyjadrujúce ich počet. Viz. príklady značenia miest na obrázku 3.2.

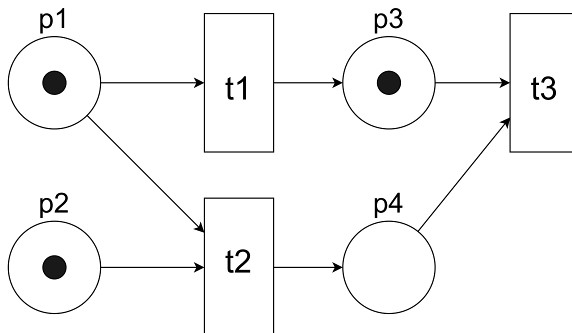


Obr. 3.2: Príklady značenia miest

3.1.3 Vykonanie prechodu

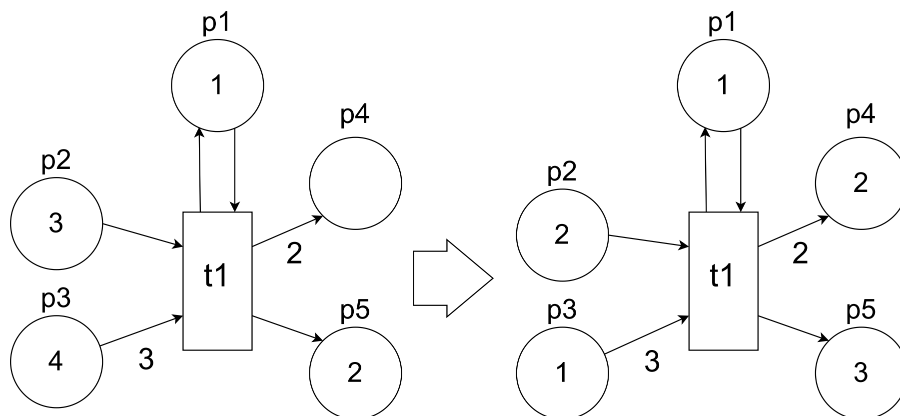
Zmena stavu v sieti sa deje vykonaním prechodu a je vyjadrená zmenou značenia v sieti. Nie každý prechod sa môže kedykoľvek vykonať. Musia byť splnené všetky podmienky pre jeho vykonanie.

Prechod je možné vykonať práve vtedy, ak za každú hranu, ktorá vedie do neho z nejakého miesta existuje v tom mieste aspoň jedna značka. To musí platiť pre všetky tieto miesta zároveň. V jednom momente môže byť vykonateľných viacero prechodov a to, že sa môže vykonať ktorýkoľvek z nich sa používa pre modelovanie paralelného a nedeterministického chovania. Nie je definované, ktorý by sa mal vykonať prednostne alebo v akom poradí. Na obrázku 3.3 je sieť s tromi prechodmi (t1, t2, t3) a štyrmi miestami (p1, p2, p3, p4). Prechody t1 a t2 sú vykonateľné, pretože t2 vyžaduje jednu značku v mieste p1 a jednu v p2 čo je splnené. Prechod t1 potrebuje len jednu značku z miesta p1, takže to je tiež splnené. t3 prechod však vykonateľný v tomto stave nie je, pretože síce má značku v p3, ale chýba značka v mieste p4. Podmienky prechodu teda niesu splnené.



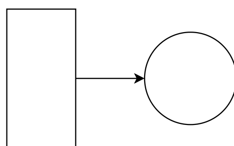
Obr. 3.3: Príklad vykonateľných a nevykonateľných prechodov

Samotné vykonanie prechodu sa uskutoční odobraním značiek za každú hranu zo vstupných miest a do výstupných miest sa uloží za každú hranu práve jedna značka.[19] Na obrázku 3.4 je príklad stavu pred a po vykonaní prechodu. Počet značiek v miestach je tu označený číslom a číslo u hrany značí, že sa jedná o viacnásobnú hranu. Prechod t1 je prepojený s piatimi miestami. Vykonaním prechodu sa počet značiek v mieste p1 nezmení, pretože je prepojené dvomi vzájomne opačnými hranami. Teda jedna značka sa odoberie a jedna vráti. Ak by tam však nebola, prechod by bol nevykonateľný. Tento prípad býva označovaný aj obojsmernou šípkou a hovorí sa potom o testovacej hrane. Z miesta p2 je odobraná jedna značka, ale keďže tam boli tri tak jedna ostáva. Z p3 sú odobrané tri značky pretože je spojené s prechodom trojnásobnou hranou a do miest p4 a p5 sú pridané dve a jedna značka.



Obr. 3.4: Príklad vykonania prechodu

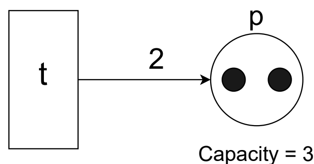
Je vhodné zdôrazniť, že počet značiek odobraných a pridaných sa nemusí rovnať. To umožňuje vytvárať napríklad generátory značiek, ktorých príklad je možné vidieť na obrázku 3.5. Prechod nemá žiadne podmienky, teda je možné ho vykonávať neustále a tak pridávať do siete ďalšie a ďalšie značky. Samozrejme sa generovanie môže podmieniť napríklad testovacou hranou.



Obr. 3.5: Generátor značiek

3.1.4 Kapacita

Maximálne množstvo značiek v mieste je možné v Petriho sieťach limitovať. Robí sa to špecifikovaním kapacity miesta a tým sa zamedzuje vykonanie prechodu, ktorý by spôsobil prevýšenie daného množstva. Je možné takto lepšie vyjadriť vlastnosti modelovaného systému, alebo zlepšiť výkon modelu. V grafickej podobe sa jednoducho pridá popis kapacity. Možné použitie je na obrázku 3.6, kde sa prechod t nemôže vykonať, pretože by preplnil miesto p.

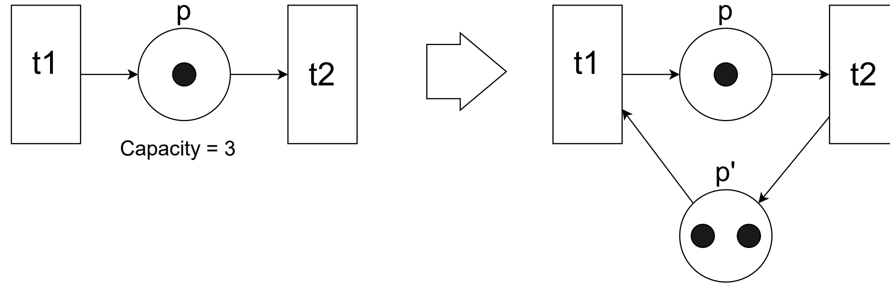


Obr. 3.6: Príklad jednoduchej siete s kapacitou.

Kapacita nie je štandardnou súčasťou Petriho sieti, ale zároveň nemení vyjadrovaciu silu, pretože každá sieť s kapacitami sa dá previesť na sieť bez kapacít pridaním komplementárnych miest. [19]

Príklad takého prevodu je znázornený na obrázku 3.7. Pre kapacitne obmedzené miesto p sa namiesto špecifikácie kapacity vytvorí doplnkové miesto, ktoré drží značky chýbajúce

do maxima. Teda ak by bola kapacita naplnená, prechod t1 sa nedá vykonať, pretože mu chýba značka v *p'*. Teda platí, že súčet značiek v mieste a jeho doplnku je vždy rovný kapacite.



Obr. 3.7: Príklad prevodu siete s kapacitou na sieť s komplementárnym miestom.

3.2 P/T Petriho siete

V tejto práci sa ďalej bude pracovať primárne s P/T Petriho sieťami definovanými následovne: P/T Petriho sieť je šestica: $PN = (P, T, F, W, C, M_0)$ [14], kde:

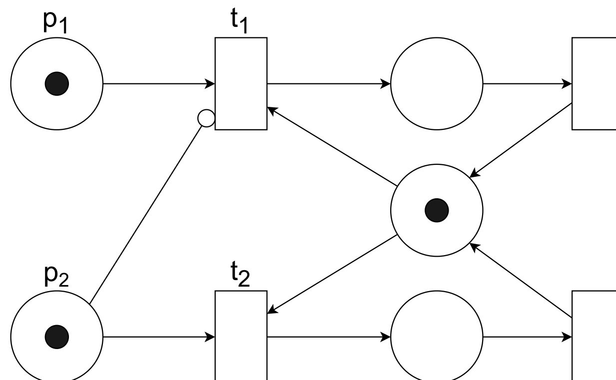
$P = \{p_1, p_2, ..., P_n\}$ je konečná množina miest.
$T = \{t_1, t_2, ..., t_m\}$ je konečná množina prechodov.
$F \subseteq (P \times T) \cup (T \times P)$ je toková relácia.
$W : F \to \{1, 2, 3, ...\}$ je ohodnotenie hrán siete udávajúce váhu hrany.
$C : P \to \{1, 2, 3, ...\} + \epsilon$ je zobrazenie určujúce kapacitu miesta.
$M_0 : P \to \{0, 1, 2, 3, ...\}$ je počiatočné značenie miest, kde platí $\forall p \in P : M_0(p) \leq C(p)$
$P \cap T = \emptyset \wedge P \cup T \neq \emptyset$

3.3 Rozšírenia Petriho sietí

Pre Petriho siete vzniklo viacero rozšírení, ktoré buď uľahčujú modelovanie komplexnejších systémov, alebo rozširujú modelovacie možnosti. Treba ale podotknúť, že tieto rozšírenia môžu viesť na zvýšenie vyjadrovacej sily Petriho sieti, čo má za následok stratu rozhodnuteľnosti o niektorých ich vlastnostiach. Ďalej budú popísané niektoré z nich.

3.3.1 Inhibítory

Prvým rozšírením, ktoré sa tu uvedie pridáva ďalší typ hrany, ktorý nazývame inhibítor. Graficky je označovaný kruhom na konci. Zatiaľ, čo bežné hrany udávajú potrebný počet značení vo vstupných miestach, inhibítory majú opačný efekt. Táto hrana zabraňuje vykonaniu prechodu, pokiaľ v pripojenom mieste je značka. [4]



Obr. 3.8: Príklad použitia inhibítoru.

Na obrázku 3.8 je možné vidieť použitie takej hrany. Prechod t_1 je možné vykonať, len vtedy, ak miesto p_2 neobsahuje žiadnu značku. Tu je možné odstrániť jedine vykonaním prechodu p_2. Efektívne to znamená, že prechod t_2 má prednosť pred t_1. Rozšírením o inhibítor hrany sa zvyšuje vyjadrovacia sila až na úroveň Turingovho stroja. [5]

3.3.2 Priorita

Ekvivalentnou alternatívou k inhibítorom je pridanie priorít prechodom. [4] Prakticky to znamená u každého prechodu číselne špecifikovať hodnotu. Následovne pri rozhodovaní, ktorý prechod sa má vykonať je zo všetkých v danom momente vykonateľných vybraný práve ten, ktorý má najvyššiu prioritnú hodnotu. Ak by však dva prechody mali rovnakú prioritu, tak je výber stále nejednoznačný.

Keďže priority sú ekvivalentnou alternatívou k inhibítorom, tak sa dá sieť z obrázku 3.8 previesť na sieť s prioritami tak, že sa odstráni inhibítor a u prechodu t_2 sa určí priorita vyššia ako u t_1. Vyjadrovacia sila, takisto ako aj množina analyzovateľných vlastností u týchto rozšírení sú rovnaké.

3.3.3 Čas

Existuje niekoľko rozšírení, ktoré pridávajú do Petriho sieti možnosť modelovať vlastnosti systému v čase. Toto je dôležité pri modelovaní systémov reálneho sveta tak ako aj výpočtových systémov. Príkladmi môže byť napríklad čas výroby súčiastky v továrni, časované svetelné semafory na dopravnej križovatke alebo modelovanie komunikačných protokolov na sieti.

Medzi rozšírenia Petriho sieti pracujúce s časom patria napríklad následovné:

- **Časované Petriho siete** (Timed Petri Nets) [18] je rozšírenie, ktoré ku každému prechodu pridáva dobu, ktorú prechod trvá, s tým, že prechod sa vykoná ihneď akonáhle je to možné. Prakticky to znamená, že zo siete sa najprv odoberú značky dané vstupnými podmienkami prechodu a až po uplynutí doby sa značky pridajú do výstupných miest. Medzi praktické využitie tohto rozšírenia patrí napríklad analýza výkonu systému.

- **Časové Petriho siete** (Time Petri nets) [10] je obecnejšia varianta, ktorá pridáva ku každému prechodu dve hodnoty. $a \leq 0$ špecifikuje dobu, po ktorú musí byť prechod pripravený na vykonanie, teda dobu, po ktorú boli splnené všetky vstupné podmienky

a hodnotu $0 \leq b \leq \infty$ udávajúci maximálny čas ktorý môže uplynúť, než sa prechod vykoná. Zároveň musí platiť, že $a \leq b$ a teda ide o interval (a, b) udávajúci dĺžku vykonania prechodu. Malo by byť jasné, že predchádzajúce rozšírenie časovaných Petriho sieti sa vie pokryť volením intervalu, kde a je rovnaké ako b.

- **Petriho siete s časovanými hranami.** (Timed-arc Petri nets)[8] Varianta, kde je časový interval špecifikovaný pre jednotlivé hrany.

3.3.4 Stochastické petriho siete

Vo svete je mnoho náhodných javov, ktoré je potreba modelovať. Táto skutočnosť sa odráža aj v rámci výpočtových systémov. Napríklad, pri modelovaní správania sa objektov v sieti, kde iba časť systému (server) je pod kontrolou a chovanie ďalších častí (klientov) môže byť často nepredvídateľné.

Formalizmus Petriho sieti sám o sebe umožňuje modelovanie nedeterministického chovania. Toto rozšírenie ale pridáva, okrem toho, možnosť zahrnúť pravdepodobnosť jednotlivých akcií.

3.3.5 Farebné Petriho siete

Farebné Petriho siete (Colored petri nets) pridávajú možnosť rozlišovať typ značky. Značka nesie informáciu o svojom typu (farbe).

Kapitola 4

Analýzy Petriho sieti

Petriho siete ako modelovací nástroj pre model checking má sadu vlastností, ktoré možno skontrolovať a spôsoby ako ich analyzovať. V tejto kapitole je uvedená podstata a princíp týchto vlastností a analýz.

4.1 Vlastnosti

Modely Petriho sieti majú niekoľko obecných rozhodnuteľných vlastností, ktoré je možno skontrolovať. Z týchto vlastností potom možno odvodiť užitočné informácie, ako napríklad neexistencia uviaznutia.

4.1.1 Bezpečnosť (Safeness)

V Petriho sietiach sa bezpečnosť viaže ku miestu. Miesto je bezpečné práve vtedy, keď počet značení v ňom nikdy nemôže byť viac ako jeden. Petriho sieť je považovaná za bezpečnú, ak všetky miesta sú bezpečné. [4]

4.1.2 Obmedzenosť (Boundness)

Obmedzenosť je vlastnosť, ktorá generalizuje bezpečnosť. Miesto je obmedzené na maximálny počet značení k ak počet značiek v danom mieste nikdy nepresiahne k a Petriho sieť je obmedzená na k, ak všetky miesta sú obmedzené na k. [4] Sieť obmedzená na k, je automaticky obmedzená na každé $l \leq k$. [19]

Malo by byť zrejmé, že ak sa baví o obmedzení, kde k je jedna, tak je to totožné s bezpečnosťou.

4.1.3 Živosť (Liveness)

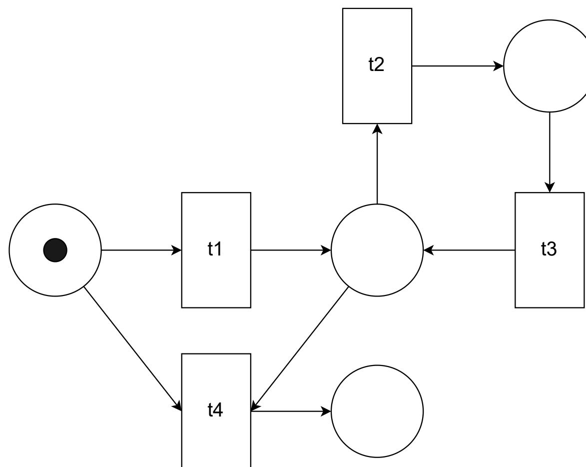
Viaže sa k prechodom. Prechod je považovaný za živý, keď je z každého značenia siete možné dostať sa do stavu, v ktorom je prechod vykonateľný. Zasa, tak ako aj iné vlastnosti, aj táto platí pre Petriho sieť, ak platí pre všetky jej prechody a hovoríme o živej sieti. Živosť je dôležitá vlastnosť, pretože ak je splnená, tak to dokazuje neprítomnosť uviaznutia.

Uviaznutie (deadlock) je situácia, kde systém nie je schopný pokračovať ďalej. Charakteristickou príčinou býva problém, kde dva alebo viac procesov vyžaduje viac zdrojov. Zaberú si ich každý časť a čakajú než sa uvoľnia zvyšné. Keďže žiadny proces nie je schopný pokračovať, tak sa systém zasekne a hovoríme o uviaznutí.

V literatúre sa uvádzajú ďalšie, slabšie úrovne živosti, ktoré už uviaznutie nevylučujú. [19] Patria sem napríklad následovné:

- Čiastočne živý prechod. Ak existuje cesta, ako vykonať prechod z počiatočného stavu, ale nie z každého možného. Na obrázku 4.1 je prechod *t1* príkladom takej situácie. Na začiatku je ako jediný vykonateľný. Potom ako sa už raz vykoná to nie je možné zopakovať.

- Mŕtvy prechod. Ak neexistuje žiadna cesta ako vykonať prechod z počiatočného stavu. Teda v systéme/modeli je chyba alebo je prechod redundantný. Prechod *t4* na obrázku 4.1 je takým prechodom. Sieť má len jednu značku, ktorá ňou putuje a *t4* vyžaduje dve, takže sa nikdy nemôže vykonať.



Obr. 4.1: Príklad siete s prechodmi rôznej úrovne živosti.

4.1.4 Konzervatívnosť (Conservation)

O konzervatívnej Petriho sieti sa hovorí, ak pre každé dosiahnuteľné značenie je súčet značiek vo všetkých miestach siete rovnaký ako súčet značiek vo všetkých miestach v počiatočnom stave. [4] To znamená, že celkový počet značiek v sieti sa nikdy nemení a zároveň, že každý prechod odoberá rovnaký počet značiek zo siete ako do nej pridáva.

Ako príklad, kedy je táto vlastnosť užitočná sa dajú brať modeli, ktoré reprezentujú zdroje systému ako značky. Potom nám táto vlastnosť zaručuje, že sa nové zdroje nepridávajú a ani neodoberajú zo systému. Príklad konzervatívnej siete je možné vidieť na obrázku 4.2.

4.2 Problémy

Vlastnosti Petriho sieti uvedené vyššie platia vtedy, ak platí, že neexistuje cesta ako sa dostať do zlého značenia. Teda takého, ktorý by tu vlastnosť porušoval.

4.2.1 Problém dosiahnuteľnosti

Problém zaoberajúci sa otázkou existencie sekvencie vykonania prechodov pre dosiahnutie špecifikovaného značenia z počiatočného značenia sa nazýva problém dosiahnuteľnosti (Reachability problem).

Značenie *M* je dosiahnuteľné práve vtedy, ak existuje postupnosť vykonania prechodov z počiatočného značenia M_0, ktoré keď sa vykoná, tak nastane značenie *M*. [14]

Dosiahnuteľnosť je rozhodnuteľný problém z triedy *NONELEMENTARY*. [6]

4.2.2 Problém pokryteľnosti

Značenie *M* je pokryté značením *M'* vtedy, ak *M'* má v každom mieste rovnako alebo viac značiek ako *M*.

Problém pokryteľnosti je podobný problému dosiahnuteľnosti, ale nemusí byt dosiahnuté presne značenie M, ale akékoľvek značenie, ktoré pokrýva *M*. [15] Znamená to okrem iného, že je možné takto vyšetrovať len časť siete, pretože v miestach, ktoré nie sú v danom momente zaujímavé, sa zvolí v cieľovom značení počet značiek ako nula a to je vždy pokryté. Rovnako ako dosiahnuteľnosť sa jedná o rozhodnuteľný problém, ale spadá do triedy EXPSPACE-complete. [7]

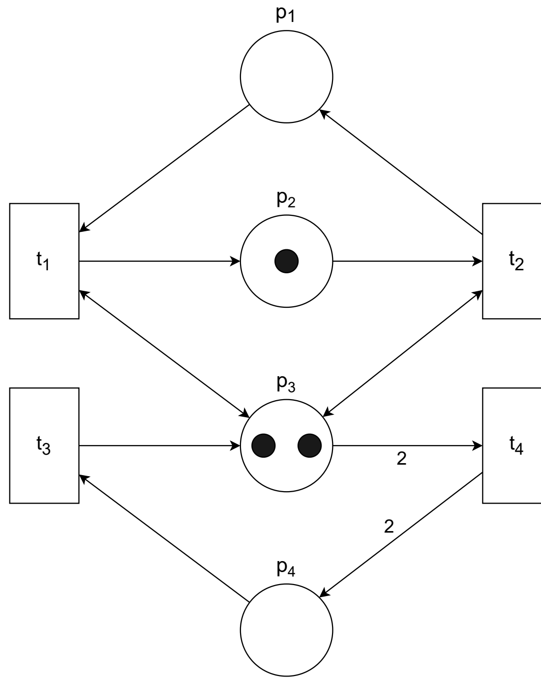
4.3 Techniky pre analýzu

S ohľadom na nekonečný stavový priestor P/T Petriho sietí, brute-force analýzy často nefungujú. Aj cez to existuje viacero analyzačných techník ako overovať vlastnosti, či už vyššie uvedených alebo iných. V tejto sekcii bude popísaných niekoľko základných z nich, ktoré sú súčasťou nástroja Netlab.

4.3.1 Maticové analýzy

Prvá technika, ktorá tu bude uvedená používa maticovú reprezentáciu siete pre odvodenie jej vlastností.

Vstupné a výstupné funkcie Petriho siete sa dajú reprezentovať za použitia pozitívnej a negatívnej matice. Pozitívna matica obsahuje informácie o pridaní značiek do miest po vykonaní prechodu a negatívna matica obsahuje informácie o odobraní značiek z miest po vykonaní prechodu. Stĺpce takejto matice zastupujú prechody a riadky zastupujú miesta.



Obr. 4.2: Príklad konzervatívnej siete.

Pre príklad Petriho siete na obrázku 4.2 pozitívna matica M^+ vyzerá následovne:

$$N^+ = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \tag{4.1}$$

Prvý stĺpec matice reprezentuje pridané značky prechodom t_1, kde pridá do miest po rade p_1, p_2, p_3, p_4 počet značiek $0, 1, 1, 0$. Negatívna matica N^- vyzerá takto:

$$N^- = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & -1 & 0 & -2 \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{4.2}$$

Negatívna matica obdobne ako pozitívna vyjadruje zmenu v miestach. Tu je možné vidieť, že prechod t_1 odoberá z miest p_1 a p_3 po jednej značke.

Lineárnou kombináciou pozitívnej a negatívnej matice vzniká matica $N = N^+ + N^-$. Označovaná ako matica zmien má pre sieť 4.2 tento tvar:

$$N = \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & -1 & 2 \end{bmatrix} \tag{4.3}$$

Matica zmien udáva zmenu značenia v sieti po vykonaní prechodov. Nereprezentuje všetky hrany, pretože ako možno vidieť z matice 4.3, tak nenesie informáciu o testovacích hranách.

Z matice 4.3 je možno overiť konzervatívnosť. Pretože pre všetky stĺpce platí, že súčet jeho prvkov je nula. Keďže stĺpec reprezentuje zmeny prechodu, tak po vykonaní prechodu je odobraných rovnaký počet prvkov ako pridaných a celkový počet značiek v sieti sa nemení. [15]

Matica reprezentujúca všetky hrany, teda všetky vzťahy medzi miestami a prechodmi sa nazýva toková matica. Tu sa získa tak, že namiesto sčítania prvkov pozitívnej a negatívnej matice sa z nich vytvoria dvojice. Pre sieť z obrázka 4.2 má toková matica tvar:

$$\begin{bmatrix} (1,0) & (0,1) & (0,0) & (0,0) \\ (0,1) & (1,0) & (0,0) & (0,0) \\ (1,1) & (1,1) & (0,1) & (2,0) \\ (0,0) & (0,0) & (1,0) & (0,2) \end{bmatrix} \tag{4.4}$$

Invariant je tvrdenie, ktoré ostáva konštantné po celú dobu behu systému. V oblasti Petriho sieti za pomoci matice zmien je možné vytvoriť dva typy invariantov.

T-invariant sa zostaví riešením sústavy lineárnych rovníc $N.x = 0$, kde N je matica zmien a x je vektor udávajúci, že ak sa každý prechod t vykoná práve *x[t]-krát*, tak sa sieť dostane do rovnakého stavu. Táto sústava rovníc pre maticu zmien siete 4.2 má dve riešenia.

$$\begin{array}{c|cc} t_1 & 1 & 0 \\ t_2 & 1 & 0 \\ t_3 & 0 & 2 \\ t_4 & 0 & 1 \end{array}$$

P-invariant sa získa taktiež riešením sústavy lineárnych rovníc, ale použije sa transponovaná matica zmien: $N^T.x = 0$

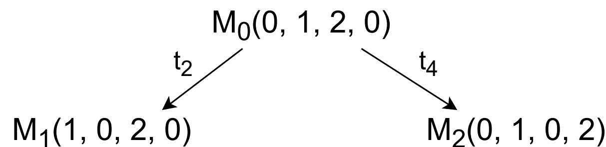
$$\begin{array}{c|cc} p_1 & 1 & 0 \\ p_2 & 1 & 0 \\ p_3 & 0 & 1 \\ p_4 & 0 & 1 \end{array}$$

4.3.2 Strom dosiahnuteľných značení

Ďalším spôsobom analýzy je strom dosiahnuteľných značení. Jedná sa o priame riešenie problému dosiahnuteľnosti pomocou hrubej sily.

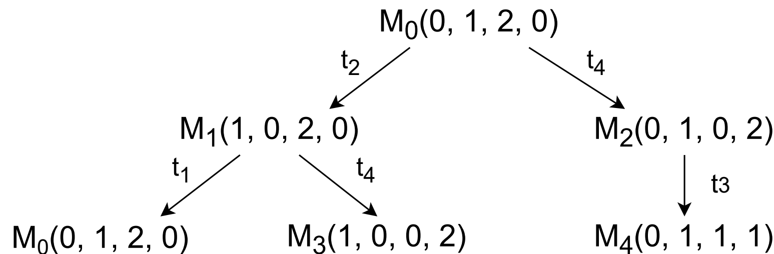
Princíp spočíva v prehľadávaní stavového priestoru. Za stav sa považuje značenie. Teda počiatočný stav je počiatočné značenie, ktoré tu bude označované ako postupnosť čísel, predstavujúce aktuálne počty značení v jednotlivých miestach. Počiatočný stav M_0 siete 4.2 sa teda označí ako $M_0(0, 1, 2, 0)$. To znamená nula značiek v mieste p_1, p_4, jedna v p_2 a dve značky v mieste p_3.

Postupuje sa tak, že koreňom stromu je počiatočný stav. Za každý vykonateľný prechod sa pridá podstrom so značením po aplikovaní toho prechodu. Z počiatočného stavu M_0 siete 4.2 sú vykonateľné dva prechody t_2 a t_4. Po prevedení tejto operácie sa vytvorí strom s dvomi listovými uzlami 4.3.



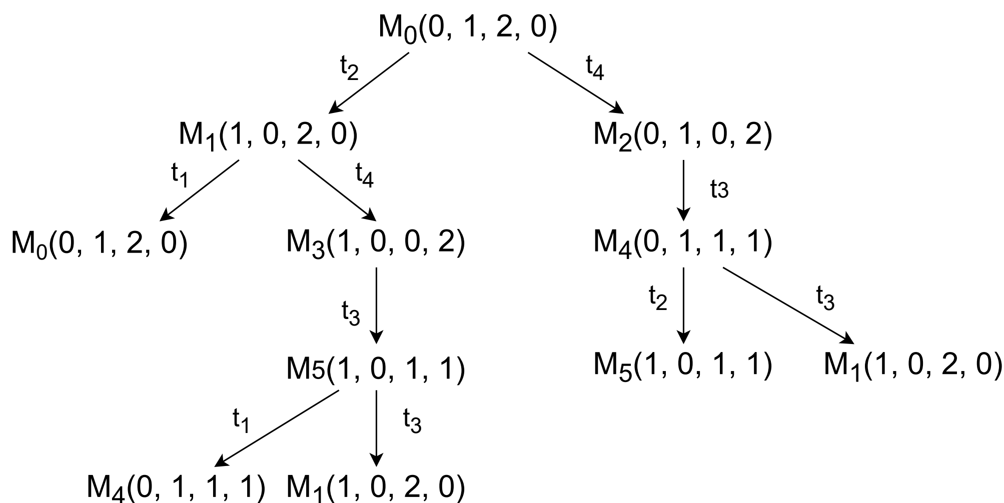
Obr. 4.3: Prvý krok tvorby stromu dosiahnuteľných značení

V ďalšom kroku sa postup opakuje pre každý listový uzol. Vzniká stav 4.4.



Obr. 4.4: Druhý krok tvorby stromu dosiahnuteľných značení

Zo značenia M1 sa vykonaním prechodu t1 dostalo do už generovaného značenia m0. To znamená cyklus. To znamená, že rozgenerovanie tohto podstromu nepomôže k nájdeniu ďalších značení. Preto sa nad ním už nebude vykonávať žiadna operácia. Postup sa opakuje dokiaľ je možné generovať nové stavy.



Obr. 4.5: Strom dosiahnuteľných značení siete 4.2

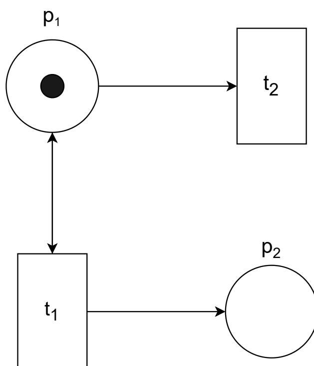
Výsledný strom dosiahnuteľných značení 4.5 zahrňuje šesť unikátnych uzlov, dávajúcich šesť dosiahnuteľných značení siete z počiatočného značenia.

Takto vytvorený strom umožňuje jednoducho rozhodnúť o dosiahnuteľnosti akéhokoľvek značenia alebo vlastnosti (4.1) siete.

- **Bezpečnosť** je tu jednoznačne porušená, pretože napríklad značenia M_0 a M_2 majú v miestach p_3 a p_4 po dvoch značkách.

- Sieť je **obmedzená** na maximum dvoch značení. Pričom miesta p_1 a p_2 len na jedno.
- **Živosť** je možné ukázať tak, že všetky prechody boli použité a z každého značenia je možné dostať sa späť do počiatočného, takže je možné ich opakovať.
- Sieť je **konzervatívna**, pretože súčet značiek v každom dosiahnuteľnom značení je práve tri.

Problém nastáva ak sieť nie je obmedzená. Príklad takej siete je na obrázku 4.6.

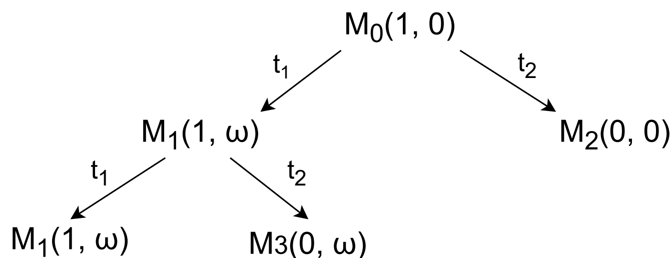


Obr. 4.6: Príklad Petriho siete, ktorá nie je obmedzená

U tejto sieti počet značiek v mieste p_2 môže narastať teoreticky do nekonečna a tvorba stromu, tak ako bolo uvedené vyššie by sa nikdy nezastavila. Na vyriešenie tohto problému sa uvádza špeciálny symbol ω a o strome vytvorenom s týmto symbolom sa hovorí ako o strome pokryteľnosti. Symbol ω predstavuje ľubovolný počet značiek a dá sa o ňom uvažovať ako o nekonečnu. Pre ľubovolnú konštantu c a symbol ω platí 4.5.

$$\begin{aligned} \omega + a &= \omega \\ \omega - a &= \omega \\ a &< \omega \\ \omega &\leq \omega \end{aligned} \tag{4.5}$$

Keď sa vygeneruje novy uzol x a není duplicitný s ďalším už existujúcim a existuje iný uzol y, ktorý je jeho predchodca na ceste z počiatočného značenia, potom ak x pokrýva y, tak zložky značenia x, ktoré obsahujú viac značení ako v x sa prepíšu na omega. [15]



Obr. 4.7: Strom dosiahnuteľných značení siete 4.6

S využitím ω sa dá pre sieť 4.6 zostrojiť konečný strom 4.7. Z počiatočného značenia $M_0(1,0)$ sa vykonaním prechodu t_1 pridá do p_2 značka a vytvorí sa nasledovník M_1 so značením $(1,1)$. Pretože M_1 pokrýva M_0 a je jeho predchodcom, tak sa značenie miesta p_2 nahradí za ω.

ω limituje rozhodovacie vlastnosti stromu dosiahnuteľných značení z dôvodu straty informácie. Preto ho nie je možné vo všeobecnosti použiť pre riešenie problému dosiahnuteľnosti a živosti. Pre ostatné tu uvedené vlastnosti bezpečnosť, obmedzenosť, konzervatívnosť a pokryteľnosť sa použiť dá. [15]

4.3.3 Spätná analýza pokryteľnosti

Spätná analýza je metóda pre overenie pokryteľnosti stavu. Na rozdiel od predchádzajúcich metód berie na vstupe okrem počiatočného značenia a siete samotnej aj stav, ktorého pokryteľnosť sa vyšetruje. [3]

Strom dosiahnuteľných značení prechádza stavy s počiatkom v počiatočnom značení. Vykonávaním prechodov sa generujú ďalšie značenia a zisťuje sa, akých stavov sa dá dosiahnuť a pokryť. Spätná analýza ide opačnou cestou. Vyberie sa stav, ktorého pokryteľnosť sa zisťuje, často sa jedná práve o zlý stav, ktorým je snaha sa vyhnúť, napríklad značenie udávajúce uviaznutie. Prechody sa generujú inverzným vykonaním prechodov s počiatkom v cieľovom stave a so snahou dostať sa do značenia pokrytého počiatočným stavom. Algoritmus tak končí buď nájdením cesty z počiatočného stavu k cieľovému, alebo vylúčením jeho pokryteľnosti.

S ohľadom na EXPTIME úplnosť problému pokrytia je časová zložitosť pre najhorší prípad taktiež EXPTIME. Avšak v priemernom prípade často dosahuje lepšiu výkonnosť ako strom pokrytia.

Algoritmus pozostáva z prehľadávania stavového priestoru a dal by sa zapísať nasledovne:

1. Inicializuj množiny *Open* obsahujúci stavy na spracovanie a množiny *Closed* pre už spracované stavy. Do *Open* vlož cieľový stav značenia.
2. Ak je množina *Open* prázdna, tak ukonči prehľadávanie **neúspechom**.
3. Vyber stav S z množiny *Open*.
4. Ak je stav S pokrytý počiatočným stavom, tak ukonči prehľadávanie **úspechom**.
5. Ak stav S pokrýva aspoň jeden stav z množiny *Closed*, tak pokračuj krokom 2.
6. Pre každý prechod $t \in T$ vytvor stav s minimálnym značením, z ktorého sa vykonaním prechodu t dostane do stavu pokrývajúceho S. Tento stav vlož do množiny *Open*.
7. Odstráň všetky stavy z množiny *Closed* pokrývajúce stav S.
8. Vlož stav S do množiny *Closed* a pokračuj krokom 2.

Kapitola 5

MFC framework

Microsoft Foundation Class skrátene MFC je framework pre vývoj aplikácii pre platformu Windows desktop. Framework bol prvý krát uvedený v roku 1992.

Cieľom MFC je zjednodušiť vývoj komplexných Windows aplikácii a na to poskytuje objektovo orientovaný wrapper pre Win32 a COM API. [13]

Vstupným bodom programu je funkcia `WinMain`, ktorá vytvorí okno a spracováva správy. Spracovávanie správ je cyklus, ktorý sa ukončí spracovaním správy `WM_QUIT`. [17] Táto správa môže byť vyvolaná napríklad kliknutím na X v systémovom menu okna alebo je vyvolaná aplikáciou samotnou. Následovne sa aplikácia ukončí. Táto časť je súčasťou MFC a programátor do nej typicky nezasahuje.

MFC definuje sadu tried pre reprezentáciu hlavných častí aplikácie ako napríklad dokument, rámec, zobrazenie, okno. Programovanie týchto častí znamená podediť z týchto tried a novovytvorenú triedu rozšíriť o špecifikované vlastnosti aplikácie. Dedené triedy obsahujú viacero metód ako pre komunikáciu medzi sebou, tak i implementáciu často potrebných rutín, ktoré je možno volať. Rozšírenie preto väčšinou znamená ich *override* a pridanie vlastnej funkcionality. MFC triedy už sami o sebe nie sú bezstavové a obsahujú viacero členských premenných. Prístup ku všetkým funkciám sa získa importovaním *afxwin.h*.

Framework sa taktiež snaží zjednodušovať a zároveň unifikovať niektoré často opakované rutiny a dialógy, ako je napríklad vybehnutie dialógu, či chce užívateľ uložiť vykonané zmeny, a preto už implementuje takéto prípady. [17]

V tejto kapitole budú ďalej popísané základné koncepty a architektúra.

5.1 Document/View architektúra

Document/View architektúra je dizajn aplikácie, ktorá rozdeľuje zodpovednosti aplikácie medzi dokument (document object) držiaci dáta a zobrazenie (view object) zobrazujúce dáta. Dokument a zobrazenie spolupracujú na interakcii s užívateľom. [17]

Dva typy *document/view* aplikácií sú podporované MFC framworkom:

- **SDI** (Single document interface). Maximálne jeden otvorený dokument. Pre malé aplikácie, kde práca na viacerých dokumentoch zároveň nedáva zmysel.
- **MDI** (Multiple document interface). Umožňuje mať zároveň otvorené viacero dokumentov. To umožňuje užívateľovi ich editovať naraz, obzvlášť vhodné, keď na sebe dokumenty závisia.

5.1.1 Aplikácia

Každá aplikácia má práve jeden objekt, nikdy nie viac, ktorý predstavuje celú aplikáciu. [17] Jedná sa o inštanciu triedy, ktorá dedí z `CWinApp` a musí byť definovaná ako globálna premenná. Týmto sa zaručí, že je objekt inicializovaný na úplnom začiatku aplikácie. Objekt žije po celú dobu behu aplikácie.

Trieda `CWinApp` definuje už spomínanú metódu `WinMain`, v ktorej beží cyklus pre spracovanie správ. Okrem toho nesie niektoré základné atribúty, ako je meno aplikácie a metódy, napríklad pre inicializáciu aplikácie alebo otváranie dokumentov, ktoré je možné prekryť (*override*).

5.1.2 Okno

Okno aplikácie je v MFC reprezentované triedou `CWnd` a každá aplikácia s grafickým užívateľským rozhraním musí mať aspoň jedno. Používajú sa ale skorej potomci tejto triedy a to `CFrameWnd`, `CMDIFrameWnd`, alebo `CDialog` [12].

Cieľom tohto objektu je zapúzdrovať funkcionalitu, ako napríklad vytvorenie okna, jeho vykresľovanie, rozloženie a spravovanie ďalších podokien.

5.1.3 Dokument

Dokument uchováva dáta aplikácie. Inštancia tohto objektu sa vytvorí z triedy dediacej triedu `CDocument`. [17] Dokument okrem uchovávania dát je aj práve ten kto nimi manipuluje.

Definuje verejnú metódu, prípadne metódy pre sprístupnenie dát pre zobrazenia(views). Tých môže mať viacero. Opačne to však neplatí. Zobrazenie môže byť spojené len s jedným dokumentom, ale dokument môže byť spojený s viacerými zobrazeniami. Odkazy na zobrazenia si uchováva v liste, ktorý sa dá získať pomocou metódy `GetFirstViewPosition` a iterovať ním použitím `GetNextView`.

Taktiež treba podotknúť, že `CDocument` je deklarovaný ako `friend` triedy CView, takže je možné pristupovať k `private` častiam zobrazenia.

5.1.4 Zobrazenie

Zobrazenie (View) objekt predstavuje grafické užívateľské rozhranie aplikácie. Implementácia *view* v aplikácii by mala dediť z `CView`, alebo nejakou jeho podtriedou. Napríklad `CScrollView`, `CEditView` alebo `CListView`.

Tento objekt má okrem zobrazovania dát ešte jednu úlohu. Prekladať užívateľský vstup na príkazy (commands), ktoré predáva na spracovanie ďalej. [17]

5.2 Správy

Správy sú fundamentálnym konceptom, ktorý slúži aplikácii pre získanie informácie o udalosti vyvolanej užívateľskou interakciou a na základe tejto správy reagovať.

Každá správa nesie štyri parametre:

- **Identifikátor okna**, ktorému je správa adresovaná. Jedná sa často o unikátny odkaz na štruktúru uchovávajúcu dáta okna.

- **Identifikátor správy**, ktorý definuje typ správy. Číselná hodnota v zdrojovom kóde definovaná pomocou makra. Napr. `WM_KEYDOWN` identifikujúci správu o stlačení klávesy.

- `wParam` a `lParam` uchovávajú ďalšie dáta potrebné pre spracovanie správy. Napr. ktorá klávesa bola stlačená.

MFC obsahuje celú radu predom definovaných správ. [17] To ale nie sú všetky správy, ktoré aplikácia spracováva. Aplikácia si môže definovať správy vlastné, ako napríklad stlačenie špecifického tlačidla grafického rozhrania alebo správy o uplynutí časovača. Teda správa môže byť vyvolaná okrem interakcie užívateľom, aj aplikáciou alebo operačným systémom.

5.2.1 Mapovanie správ

Mapovanie správ (message mapping) je spôsob pridelenia k správe reakciu v podobe vyvolania metódy. Trieda implementujúca reakcie na správy a mapovanie je definované pre každé okno aplikácie zvlášť.

Keď oknu príde správa, tak sa pozrie do svojej mapy správ a ak tam nájde metódu pridelenú k danému typu správy, tak tu metódu zavolá. V opačnom prípade správu ignoruje.

Mapu je možné definovať pomocou makier `BEGIN_MESSAGE_MAP`, `END_MESSAGE_MAP` a využitia ďalších makier pre špecifikovanie dvojíc správa-metóda. Príklad deklarovania takejto mapy pre dialóg okno je možné vidieť na príklade 5.1. Každá metóda, ktorá slúži ako reakcia na správu má na začiatku deklarácie uvedené makro `afx_msg`. To je nezbytné pre kompiláciu.

```
class CMyDialog : public CDialog
{
protected:
    virtual void DoDataExchange(CDataExchange* pDX);

// Message Map
protected:
    afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnMyCustomMessage();
    afx_msg void OnClipboardChange(UINT nID);
};

BEGIN_MESSAGE_MAP(CMyDialog, CDialog)
    ON_WM_KEYDOWN()
    ON_COMMAND(ID_MY_CUSTOM_MESSAGE, OnMyCustomMessage)
    ON_COMMAND_RANGE(ID_EDIT_COPY, ID_EDIT_PASTE, OnClipboardChange)
END_MESSAGE_MAP()
```

Výpis 5.1: Príklad mapy správ

Na príklade 5.1 je možné vidieť viacero typov mapovania. MFC však obsahuje celú radu ďalších.

- `ON_WM_KEYDOWN` je makro definujúce reakciu na MFC typ správy pre stlačenie klávesy a ako reakcia je vyvolaná metóda s názvom `OnKeyDown`, ktorej meno a parametre sú predom dané frameworkom.

- `ON_COMMAND` je makro pre generické mapovanie, používané hlavne pre užívateľsky definované typy správ. Má dva argumenty. Prvým je identifikátor správy. V tomto prípade taktiež definovaný makrom `ID_MY_CUSTOM_MESSAGE`. Druhým je identifikátor metódy, ktorá je volaná ako reakcia, definovaná vyššie. (`OnMyCustomMessage`)

- `ON_COMMAND_RANGE` je makro umožňujúce jednoducho reagovať rovnakým spôsobom na viacero typov správ zároveň. A to špecifikovaním ID správ, ktorých sa to týka. Takže v tomto prípade je metóda `OnClipboardChange` volaná pre obe správy s identifikátormi `ID_EDIT_COPY` a `ID_EDIT_PASTE`. Do metódy je následovne predané ID správy ako parameter.

5.3 Kreslenie

MFC umožňuje vykresľovanie, alebo aj kreslenie samotné v rámci okien aplikácie. Pre tento účel sa využíva GDI (Graphics Device Interface). Ten poskytuje rozhranie pre kreslenie po obrazovke.

5.3.1 Device context

Základom pre použitie GDI je kontext zariadenia (device context). Je to dátová štruktúra, udržujúca informácie o prostredí, ako napríklad cieľové zariadenie. V tomto prípade okno. MFC poskytuje viacero tried pre prácu s kontextom zariadenia.

- `CDC` je základnou triedou pre prístup ku kontextu a ďalšie triedy pre tento účel sú jeho potomkami. Poskytuje metódy mimo iné pre vykresľovanie a prístup k niektorým pokročilejším GDI objektom (5.3.2).

- `CPaintDC` je používaný pri spracovaní správy `WM_PAINT`, ktorá býva vyvolaná operačným systémom, keď je potrebné vykresliť okno. To sa spravidla deje v situáciach, keď sa otvorí okno, zmenia rozmery okna alebo keď sa obsah okna zmení.

- `CClientDC` sa vytvára pre **mimo** obsluhy správy `WM_PAINT` a dáva aplikácii možnosť kresliť. Napríklad pre animácie.

`CDC` poskytuje sadu metód pre vykresľovanie jednoduchých útvarov a primitív. Medzi tieto patria napríklad: body, čiary, obdĺžniky, polygóny alebo elipsy. Príklady ich vykreslenia je možné vidieť na príklade 5.2

```
// Create context
CClientDC dc(this);
// Draw a green pixel
dc.SetPixel(200, 200, RGB(255, 0, 0));
// Draw a line
dc.MoveTo(50, 50);
dc.LineTo(150, 150);
// Draw a rectangle
CRect rect1(200, 50, 300, 150);
dc.Rectangle(&rect1);
// Draw polygon
```

```
POINT points[] = {{ 200, 200 },{ 250, 250 },{ 300, 200 },{ 250, 300 }};
dc.Polygon(points, 4);
// Draw a ellipse
CRect rect2(50, 200, 150, 250);
dc.Ellipse(&rect2);
```

Výpis 5.2: Príklad vykreslenia základnych útvarov pomocou CDC

5.3.2 GDI Objects

Okrem metód pre vykresľovanie základných objektov `CDC` poskytuje prístup k GDI objektom, ktoré predstavujú nástroje pre komplexnejšie operácie. Jedná sa o perá, štetce, text alebo obrázky.

- **Pero**. Reprezentované triedou `CPen` je základný nástroj slúžiaci pre kreslenie čiar rôznych tvarov, veľkostí a farby.
- **Štetec**. Trieda `CBrush` pre vypĺňanie vnútra nejakého útvaru špecifikovanou farbou a vzorom.
- **Text**. Využitím triedy `CFont` sa vykresľuje text. Rôzne typy písma, farby textu, či veľkosti môžu byť použité.
- **Obrázky** sa dajú zobraziť za pomoci `CBitmap`.

Na príklade 5.3 je ukázaný postup pre použitie štetca. Najprv sa štetec inicializuje pomocou `CreateSolidBrush` na modrú farbu. Nastaví sa ako aktívny: `CDC::SelectObject` a po vykreslení obdĺžnika je jeho obsah modrou farbou. `SelectObject` metóda vracia pointer na predchádzajúci objekt, pretože aktívny môže byť iba jeden. Preto je pointer potrebný, aby bolo možné sa vrátiť kontext do pôvodného stavu.

```
// Create a blue color brush
CBrush brush;
brush.CreateSolidBrush(RGB(0, 0, 255));
// Get device context
CDC* pDC = GetDC();
// Select the brush
CBrush* pOldBrush = pDC->SelectObject(&brush);
// Draw a rectangle using the selected brush
CRect rect(10, 10, 100, 100);
pDC->Rectangle(&rect);
```

Výpis 5.3: Príklad vykreslenia základnych útvarov pomocou CDC

Kapitola 6

Netlab

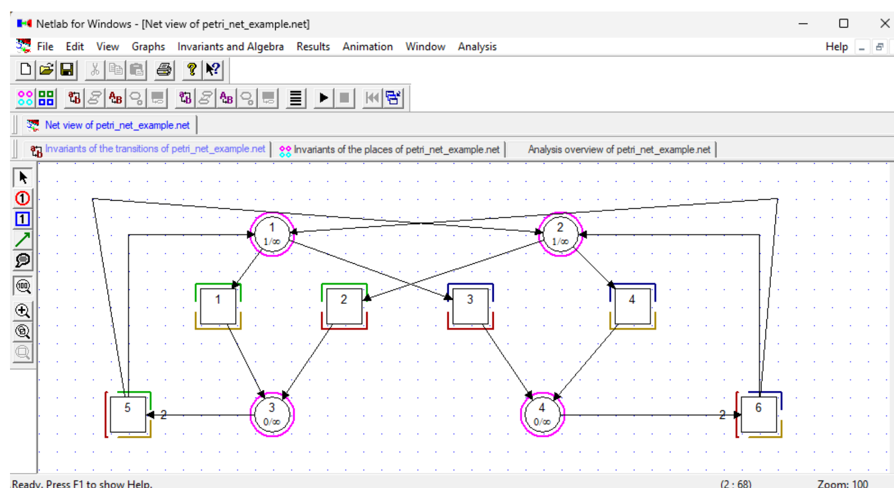
V tejto kapitole budú popísané hlavné časti aplikácie Netlab, nástroja vyvinutého na RWTH AAchen univerzite (Institute of Automatic Control) pre platformu Windows na tvorbu Petriho sieti a ich analýzu.

6.1 Nástroj Netlab

Nástroj poskytuje užívateľské rozhranie nad štruktúrou Petriho sietí a umožňuje nad ňou vykonávať radu funkcií. Medzi hlavné funkcie patria:

- Grafický editor pre tvorbu a editáciu P/T Petriho sietí
- Simulátor Petriho sieti
- Funkcie pre grafovú a invariantnú analýzu.
- Modul pre analýzu výsledkov

Nástroj mimo to zvláda prácu nad viacero sieťami zároveň, ich ukladanie a načítanie zo súborov. Jazyková podpora zahŕňa angličtinu, nemčinu a španielčinu, ktoré je možné prepínať v menu.[1]



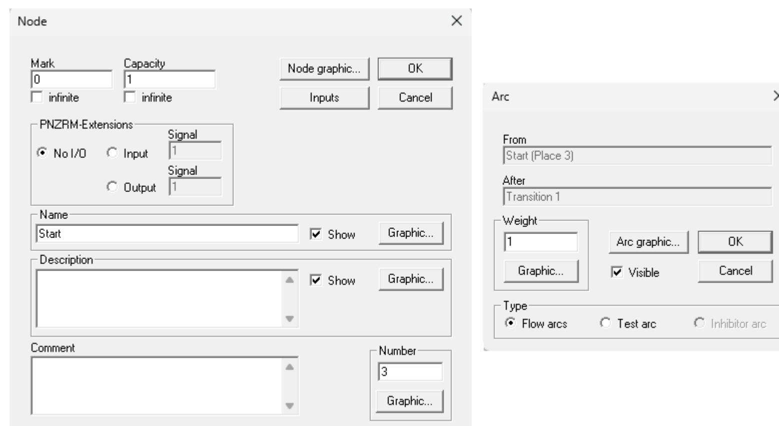
Obr. 6.1: Nástroj Netlab

6.2 Grafický editor

Tvorba a modifikácia Petriho siete je základnou funkciou tohto nástroja. Užívateľ má k dispozícii panel nástrojov, ktorý je možno vidieť na ľavej strane obrázka 6.1. Medzi nástrojmi sú miesto, prechod a hrana pre ich pridanie do siete.

Sieť možno ďalej modifikovať kliknutím na jednotlivé prvky, čo otvorí dialóg pre editáciu ich grafických vlastností ako je meno, popisok, farba alebo font. Dialóg okrem toho umožňuje aj úpravu funkčných častí prvku. Viz. dialógy na obrázku 6.2.

- Miesto je možné upraviť špecifikovaním kapacity a počiatočného značenia.
- U hrany možno špecifikovať váhu hrany a typ hrany, ktorá môže byť buď bežná toková hrana alebo testovacia hrana reprezentovaná obojsmernou šípkou.



Obr. 6.2: Dialógové okná pre úpravu vlastností siete.

6.3 Simulácia

Ďalšou podstatnou funkciou je možnosť simulovať chovanie siete pomocou vstavaného simulátora. Ten má tri tlačidlá na paneli nástrojov pre spustenie, zastavenie a reštart simulácie.

Po spustení sú zvýraznené prechody, ktoré je možné vykonať. Kliknutím na ne sa vykonajú a značky sa odoberú zo vstupných miest prechodu a uložia do výstupných miest.

6.4 Invarianty a grafová analýza

Nad vytvorenou sieťou nástroj umožňuje vykonávať analýzy.

6.4.1 Invarianty

Netlab umožňuje zo siete vypočítať P a T invarianty kliknutím na príslušné tlačidlá. Výpočet môže skončiť neúspechom ak sa invariant nepodarí zostrojiť a vyskočí iba informačný dialóg. V opačnom prípade sa otvorí okno s výsledkami v textovej podobe. Zároveň sa invarianty graficky znázornia na sieti (obrázok 6.1). Spôsob výpočtu invariantov je závislí na maticovej reprezentácii siete, ktorú Netlab taktiež sprístupňuje. Samotný výpočet je priblížený v sekcii 4.3.1.

6.4.2 Grafy

Ďalšou funkciou pre analýzu je výpočet grafu dosiahnutia a grafu pokrytia. Tieto funkcie obe vytvárajú strom dosiahnuteľných značení pomocou algoritmu uvedeného v sekcii 4.3.2, kde graf dosiahnutia použije variantu bez špeciálneho znaku ω a graf pokrytia s ním. Výstup týchto algoritmov má len textovú podobu (6.3) Znak ω je na výstupe reprezentovaný znakom „*".

```
Coverability graph of Petri net.net
M001:   0 ( 1 0 0 0 0) ---t1---> M002:   1 ( 0 1 0 0 0)
                       ---t5---> M003:   1 ( 1 0 0 0 *)
M002:   1 ( 0 1 0 0 0) ---t2---> M004:   2 ( 0 0 1 0 0)
M003:   1 ( 1 0 0 0 *) ---t1---> M005:   2 ( 0 1 0 0 *)
                       ---t5---> M003:   1 ( 1 0 0 0 *)
M004:   2 ( 0 0 1 0 0) ---t3---> M002:   1 ( 0 1 0 0 0)
M005:   2 ( 0 1 0 0 *) ---t2---> M006:   3 ( 0 0 1 0 *)
M006:   3 ( 0 0 1 0 *) ---t3---> M005:   2 ( 0 1 0 0 *)
Graph construction is complete!
```

Obr. 6.3: Textová reprezentácia grafu pokryteľnosti.

6.4.3 Výsledky analýzy

Nástroj Netlab používa vytvorené grafy a invarianty pre odvodenie vlastností siete ako obmedzenosť a živosť, bližšie popísané v sekcii 4.1 ale aj identifikuje možné uviaznutia a mŕtve prechody. Tieto výsledky sú prístupné z *„Results"* záložky hlavného menu. Taktiež sa jedná čisto o textový výstup.

6.5 Import/Export

Sieť vytvorenú nástrojom Netlab je možné uložiť pre neskoršie použitie, alebo pre použitie v inom kompatibilnom nástroji.

Aplikácia umožňuje ukladať sieť do súborov v jazyku PNML (Petri Net Markup Language). Ide o jazyk založený na XML a štandardizujúci formát v ktorom sú Petriho siete uložené.[20]

V PNML formáte je možné sieť ako exportovať, tak i importovať. Iné formáty exportu sú dokumenty *pdf* a obrázky vo formáte *png*.

Kapitola 7

Netlab implementácia a vývojové prostredie

7.1 Vývojové prostredie

Cieľom tejto časti práce je umožniť kompiláciu a vývoj nástroja Netlab za použitia aktuálne podporovaných prostriedkov a závislostí pre vývoj. Zároveň bola venovaná aktivita odstráneniu, alebo minimalizáciu závislostí pre uľahčenie vývoja.

7.1.1 Pôvodné vývojové prostredie

Zdrojové kódy aplikácie obsahovali súbor riešenia s príponou *.sln* a súbory projektov s príponou *.vcproj* indikujúce, že sa vývoj a preklad robil v prostredí Visual Studio (VS). Obsah súboru *Netlab.sln* obsahoval informáciu, že sa jedná špecificky o Visual Studio 2005 a súbory s príponou *.vcproj* informáciu, že je použitá technológia MFC uvedená v kapitole 5 a jazyk *Visual C++* s verziou 8.0.

Posledný oficiálne kompatibilný Windows operačný systém s VS 2005 je Windows Vista. Aj cez tento fakt sa ale podarilo po prekonaní niekoľko neľahkostí VS 2005 rozbehnúť na zariadení s Windows 11. To však nie je ideálne, keďže beh nie je nijako garantovaný. Preto sa rozhodlo prejsť na novšiu verziu VS, ktorá okrem toho umožní aj novší štandard jazyka C++. Pre samotný preklad je potrebné mať na vývojovom zariadení ďalšie technológie. Konkrétne Matlab a Perl. Po ich inštalácii sa preklad podaril.

7.1.2 Prechod na nové prostredie

Ako novšie vývojové prostredie sa vybralo to najnovšie možné v dobe písania tejto práce. A to Visual Studio 2022. S ohľadom na celkom veľký skok sa prejavilo niekoľko problémov, ktoré bolo potrebné vyriešiť.

Vývojárske nástroje. VS 2022 vo svojom základe neobsahuje podporu pre MFC a je potrebné ho nainštalovať cez VS inštalátor.

Migrácia projektov. VS umožňuje automatizovane migrovať staršie projekty na novšie. Nie je to však vždy bez problémov, ktoré vznikajú ak časti projektov, alebo závislostí, už nie sú podporované. S ohľadom na to sa konverziou nepodarilo prejsť jednému projektu a polovica prešla s varovaniami. Konverzia upravila súbor *Netlab.sln* a z každého projektového súboru s príponou *.vcproj* vygenerovala súbory s príponami *.vcxproj* a *.vcxproj.filters*. Ďalej bolo potrebné do týchto projektov zasiahnuť, pretože definovali minimálnu verziu

Windows, ktorú aplikácia podporuje. Definovaná bola verzia pre *Windows 2000*. Keďže tá už nie je podporovaná, tak to spôsobovalo chybu prekladu.

Visual C++ prekladač, ktorý je súčasťou VS 2022 vyžaduje pre preklad kompatibilitu minimálne štandard C++14. Prekladač prostredia Visual Studio 2005 nebol úplne kompatibilný s C++11 štandardom. Táto nekompatibilita spôsobila viacero chýb, ktoré bolo potrebné odstrániť, aby bol kód v súlade so štandardom C++14. Prevod sa však vykonal tak, aby podporoval až C++17. Chyby by sa dali rozdeliť do dvoch kategórií podľa ich vzniku.

- Neštandardná funkcionalita VS kompilátora, ktorá bola štandardizovaná. Príkladom môže byť `stdext::hash_map`, pre ktorú vznikla `std::unordered_map`, ako alternatíva v C++ štandardne. Aj keď táto funkcionalita môže byť stále použitá pomocou direktívy prekladaču, tak boli prevedené.

- Štandardná funkcionalita, ktorá bola odstránená. `std::binary_function` môže byť príklad takej funkcionality. Bola označená ako zastaralá v C++11 a úplne odstránená v C++17 bez priamej náhrady.

Najviac problémov bolo v súbore *StdString.h*, v ktorej je definovaná trieda `CStdString`. Jedná sa o viac ako štyritisíc riadkov kódu rozširujúci `std::basic_string` o zjednodušenie a zjednotenie práce s MFC `CString` triedou, COM `IStream` a štandardnými typmi. `CStdString` je používaný v širokom rozsahu v celej aplikácii. Čo uľahčuje vývoj, ale nesie to aj niekoľko nevýhod. Medzi hlavné patria:

- Trieda je závislá na Windows špecifických závislostiach, čo vytvára všetok kód, v ktorom je použitá závislý na windows.

- Jedným zo závislostí je nepriamo hlavičkový súbor `windows.h`, ktorý by sa nemal používať v kombinácii s MFC knižnicami, ktoré vracajú chybu ak sú spracované a `windows.h` je definovaný. Tento problém je obchádzaný používaním `#include` direktívy v správnom poradí a to tak, že sa vždy najprv importujú MFC knižnice a až potom *StdString.h*

7.1.3 Matlab a Perl

Zdrojové kódy obsahovali niekoľko Matlab a Perl skriptov.

Perl je použitý v dvoch súboroch a to *help.pl* a *build_search_database_win.pl*. Tieto sú používané pre zostavenie html nápovedy.

Volanie týchto skriptov bolo súčasťou prekladu aplikácie vo forme vlastného kroku, ktorý definoval postupnosť príkazov, medzi ktorými bolo volanie zmienených skriptov, pre zostavenie *netlab.chm* nápovedy. Tento súbor nie je potrebný pre ďalší preklad a aplikácia beží aj bez neho. Ak ale nie je prítomný v zložke s ostatnými binárnymi súbormi, tak pri pokuse o jeho otvorenie sa len zobrazí chybový dialóg. S ohľadom na to sa rozhodlo oddelenie tohto kroku do samostatného projektu v rámci VS riešenia s názvom *Help*, na ktorom nie je nič ďalšie závislé. Vďaka tomu je možné prekladať aplikáciu aj bez Perlu.

Matlab na druhú stranu plní úlohu sprístupnenia používať Netlab cez Simulink pomocou COM rozhrania, ktoré Netlab definuje. Samotný preklad aplikácie ale nie je na tomto nijako závislí a pretože to nie je cieľom tejto práce, tak boli tieto skripty spolu s VS projektami, v ktorých boli použité vynechané z výsledného riešenia. Tým sa závislosť na Matlabu kompletne odstránila.

7.1.4 Projekty

Výsledné riešenie po úpravách v rámci tejto práce je rozdelené na najvyššej úrovni do deviatich projektov.

- *Netlab* projekt, ktorý obsahuje väčšinu zdrojového kódu a je hlavnou častou riešenia, viac bude rozobraný v sekcii 7.3. Pôvodne výsledkom zostavenia tohto projektu bol priamo binárny súbor aplikácie. To však znemožňovalo závisieť na obsiahnutom kóde z iného projektu. Preto z neho bola oddelená minimálna časť vo forme nového VS projektu *NetlabApp*, z ktorého vzniká binárny súbor a z *Netlab* projektu sa stala statická knižnica. *Netlab* priamo závisí pre svoju kompiláciu na ďalších troch projektoch.

- *Netlab.Tests* je druhým projektom, závislým na statickej knižnici *Netlab*. Bol vytvorený pre možnosť implementovať testy s využitím testovacieho frameworku Google Test.

- *InterfaceDefinition* definuje COM rozhranie a *ProxyStup* obsahuje vygenerované rozhranie z tejto definície a produkuje súbor s príponou .dll(dynamicly-linked library)

- *CxmlLib* je tretou závislosťou *Netlab* projektu. Taktiež sa jedná o DLL a obsahuje implementáciu knižnice pre spracovávanie XML súborov použitú pre import/export operácie.

- *Netlab_de* a *Netlab_es* sú projekty produkujúce DLL pre nemeckú a španielsku jazykovú podporou.

- *Help* projekt už spomínaný vyššie pre zostavovanie súboru s nápovedou.

7.2 Kvalita kódu

Nástroj Netlab nebol viac ako desať rokov vyvíjaný. To spolu s prechodom na novšie verzie nástrojov vytvorilo technický dlh v rôznych formách. Pre uľahčenie ďalšieho vývoja bolo vykonané množstvo úprav, ktoré by to mali pomôcť zmeniť.

Prvým adresovaným problémom bol fakt, že zdrojový kód aplikácie bol písaný v nemeckom jazyku. Jednalo sa o viac ako deväťdesiat percent komentárov, identifikátorov a názvov súborov. Pre nemecky nehovoriaceho človeka to môže predstavovať značnú bariéru. Preto bolo vykonané značné úsilie v preklade kódu do anglického jazyka.

Ďalším problémom, bol fakt, že pre minimálne prvé spustenie aplikácie bolo vyžadované administrátorské oprávnenie. Investigáciou sa zistilo, že aplikácia vyžaduje pre inicializáciu zápis do systémových registrov. Toto nemusel byť vždy problém, pretože tento typ reštrikcií vznikal až v novších verziách Windows. Pre odstránenie tohto problému sa všetka práca s registrami presmerovala na užívateľské registre, ktoré administrátorské oprávnenia nevyžadujú. Toto sa podarilo vykonať vcelku jednoducho volaním metódy MFC knižnice `AfxSetPerUserRegistration` s parametrom `TRUE` počas inicializácie programu. Po tejto zmene nie sú administrátorské práva potrebné.

Okrem tohto bolo vykonané veľké množstvo malých zmien, pre zvýšenie kvality kódu. Medzi vykonané zmeny patria napríklad následovné:

- Odstránenie mŕtveho alebo zakomentovaného kódu.

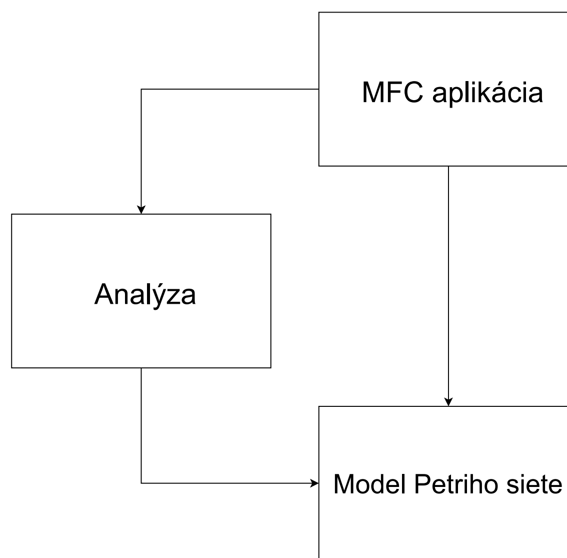
- Riešenie varovaní pri kompilácii rôzneho typu

- Aplikácia konštrukcií C++ jazyka pridaných novším štandardom.
- Odstránenie rady duplicitného kódu dekompozíciou funkcií.
- Úprava štruktúry projektu pre zjednodušenie importov

7.3 Architektúra

Netlab aplikácia sa dá sémanticky rozdeliť na tri časti. Model, analýza a MFC aplikácia. V tejto sekcii budú popísané tieto časti v stave po vykonaní zmien uvedených v časti 7.2 s cieľom priblížiť implementáciu aplikácie. Jedná sa o popis VS projektu *Netlab*.

Z obrázka 7.1 je možné vyčítať, ako sú na sebe časti závislé. V dolnej časti je model Petriho siete, s ktorým manipuluje priamo MFC aplikácia a volá analýzu nad ním.



Obr. 7.1: Časti Netlab aplikácie a ich závislosti.

7.3.1 Model

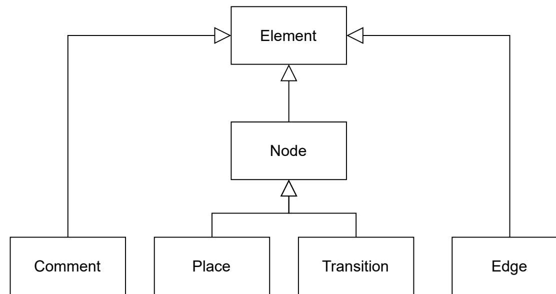
V tejto časti sa nachádzajú definície tried, ktoré interne reprezentujú štruktúru siete. Každý primitívny element Petriho siete tu má vlastnú triedu, ktorá ho reprezentuje. Pretože prvky zdieľajú spoločné operácie a rozhrania, tak je tu aplikovaná dedičnosť tried. Každá trieda je implementovaná v dvojici súborov s príponami *.h/.cpp*. Meno súboru je inak vždy totožné s menom triedy.

- `Element` je základnou triedou, z ktorej priamo či nepriamo dedia všetky prvky siete uvedené v tomto zozname. Definuje základné rozhranie, ktoré všetky prvky siete musia implementovať. Okrem toho drží identifikačné číslo a inštanciu triedy `PNML_Graphics`, obsahujúcu vykresľovanie dáta prvku, ako napríklad poloha na plátne.
- `Comment` predstavuje textové poznámky, bez akejkoľvek ďalšej funkcie. Mimo implementácie základného rozhrania `Element`, drží len informáciu o zobrazovanom texte.
- `Edge` popisuje orientované hrany siete. Nesie v prvom rade dva ukazovatele na uzly, ktoré spája. Okrem toho ale uchováva aj informácie o bodoch, cez ktoré prechádza,

typ hrany (*Flow/Test*) a váhu. K tomu definuje ešte sadu metód pre prácu nad týmito dátami.

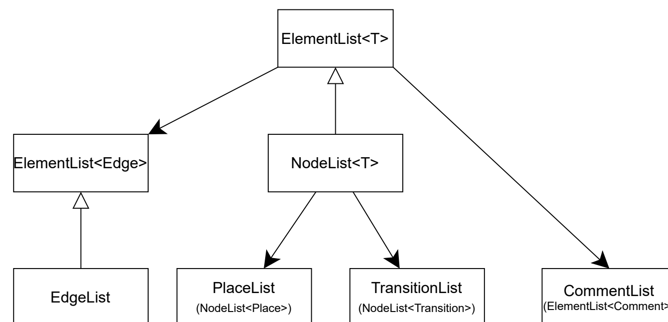
- `Node` zastupuje uzly siete. Obsahuje veľké množstvo spoločnej i špecifickej logiky pre miesta a prechody. Drží množstvo informácií od odkazov na hrany k nemu pripojených, cez značenie a kapacitu, až po extra informácie o uzle ako meno a popisok.

- `Place` a `Transition` dedia z triedy `Node`. Ten už obsahuje takmer všetky potrebné dáta. Tieto triedy pridávajú len implementáciu rozhrania triedy `Element`.



Obr. 7.2: Závislosti medzi triedami reprezentujúce prvky siete.

Závislosti medzi vyššie spomenutých tried sú znázornené na obrázku 7.2. Nad týmito triedami sú implementované kontajnery, ktoré implementujú logiku pre prácu s množinou týchto prvkov. Vzťahy tried kontajnerov možno vidieť na obrázku 7.3.



Obr. 7.3: Závislosti kontajnerov pre ukladanie zoznamov prvkov Petriho siete.

- `ElementList<T>` je šablóna triedy (template class) pre zoznam prvkov siete. Obdobne ako `Element` je základom každého prvku, tak `ElementList<T>` je základ každého zoznamu. Interne ukladá prvky za pomoci `std::list` a implementuje operácie nad zoznamom generické pre všetky prvky. Z tejto šablóny sa priamo vytvára inštancia len pre zoznam komentárov vo forme `ElementList<Comment>`. Pre ostatné prvky sú vytvorené vlastné listy dediace tento.

- `EdgeList` dedí inštanciu triednej šablóny `ElementList<Edge>` a pridáva špecifickú funkcionalitu pre hrany.

- `NodeList` nededí inštanciu šablóny, ale dedí samotnú šablónu `ElementList<T>`, takže sa tiež jedná o šablónu. Pridáva metódy pre prácu s uzlami a vytvára sa jej inštancia pre zoznam miest a zoznam prechodov.

Trieda, interne reprezentujúca celý model Petriho siete, má názov `PetriNet` a dáta o sieti jej inštancie uchovávajú vo forme štvorice listov: `PlaceList, TransitionList, EdgeList, CommentList`. Vo vyšších úrovniach riešenia sa pracuje práve s touto štruktúrou pre reprezentáciu siete.

Ďalšou dôležitou triedou je trieda `Selection`. Zabezpečuje funkcionalitu pre výber časti Petriho siete a operácie nad ňou. Inštancia tejto triedy sa z pravidla vytvára pri vybraní časti grafického modelu. `Selection` zabezpečuje aby sa vždy vybrala správna časť siete.

Základom je metóda `Select`, pre výber časti siete. Berie ako argument objekt triedy `PnRect` reprezentujúci obdĺžnik, v tomto prípade vybranú obdĺžnikovú oblasť. Nad vybranou časťou umožňuje vykonávať operácie mazania, kopírovania, presunu, otáčania a ďalších.

Jednotlivé triedy v rámci modelu implementujú aj logiku pre ich vykreslenie, preklad do a z jazyka PNML.

Vykresľovanie je implementované tak, že trieda `Element` deklaruje čisto virtuálne funkcie `Draw` a `DrawSelection`. Takže každá trieda, ktorá ju dedí a má sa inštanciovať musí tieto metódy definovať. Triedy teda definujú ako sa vykresliť. Ich implementácie sa nachádzajú v súbore *WinDraw.cpp*. Metódy berú na vstupe objekt `CDC` z MFC knižnice, do ktorého svoj objekt vykreslia. Trieda `CDC` je bližšie uvedená v sekcii 5.3.1.

Pre preklad do a z jazyka **PNML** už spomenutého v sekcii 6.5 trieda `PetriNet` definuje dvojicu metód `PNML` a `DePNML`. Takúto dvojicu definujú všetky triedy siete pre preklad ich špecifickej časti a obdobne ako u vykresľovania je aj tu takmer všetka logika implementovaná v jednom súbore *PNML.cpp*. Oddelená je len implementácia prekladu grafických vlastností prvkov a to v triede `PNML_Graphics`.

7.3.2 Analýza

Druhou časťou Netlabu je časť zodpovedná za analýzu siete. Prvky objektovo orientovaného programovania sú tu použité rozdelením každej z analýz do samostatnej triedy.

Invarianty sú implementované v súboroch *InvariantAnalysis(.h/.cpp)*. Základom je trieda `Invariant` implementujúca výpočet. Dedia ju triedy `T_Invariant` pre invariant prechodov a `P_Invariant` pre invariant miest upravujúce pre nich špecifický výpočet, ktorý sa delí len tým, že invariant miesta používa transponovanú maticu siete. Princíp výpočtu je priblížený viac v sekcii 4.3.1.

Grafová analýza pre zostrojenie grafu pokrytia je implementovaná triedou, ktorá z Petriho siete vloženej na vstupu vytvorí graf dosiahnuteľnosti. Jej inštancie držia takto vytvorený graf. Jedná sa o triedu s názvom `ReachabilityGraph`. Tvorba stromu odpovedá algoritmu popísanému v sekcii 4.3.2 bez použitia špeciálneho znaku ω. Algoritmus s ním je implementovaný triedou `CoverageGraph` upravením chovania rodiča `ReachabilityGraph`.

Vytvorený graf sa skladá z uzlov `GraphNode` a hrán `GraphEdge` uložených v zozname uzlov `GraphNodeList` a zozname hrán `GraphEdgeList`.

Netlab ukladá aj reprezentáciu grafu v redukovanej forme. Toto je zabezpečené triedou `GraphCondensation` skladajúcu uzly grafu v podobe `CondensedGraphNode`.

Analýza výsledkov je ďalej vykonávaná triedou `ResultAnalysis`, ktorá z výsledkov vyššie uvedených grafov a invariantov vyšetruje vlastnosti siete ako napríklad možnosť uviaznutia, prítomnosť mŕtvych prechodov a ďalšie.

Všetky analýzy dedia triedu `OutputBase` definujúcu rozhranie metód, ktoré sú volané aplikáciou pre získanie textovej reprezentácie výsledkov a metódu `Make`, ktorá berie ako parameter ukazovateľ na sieť z ktorej má graf alebo invariant vytvoriť.

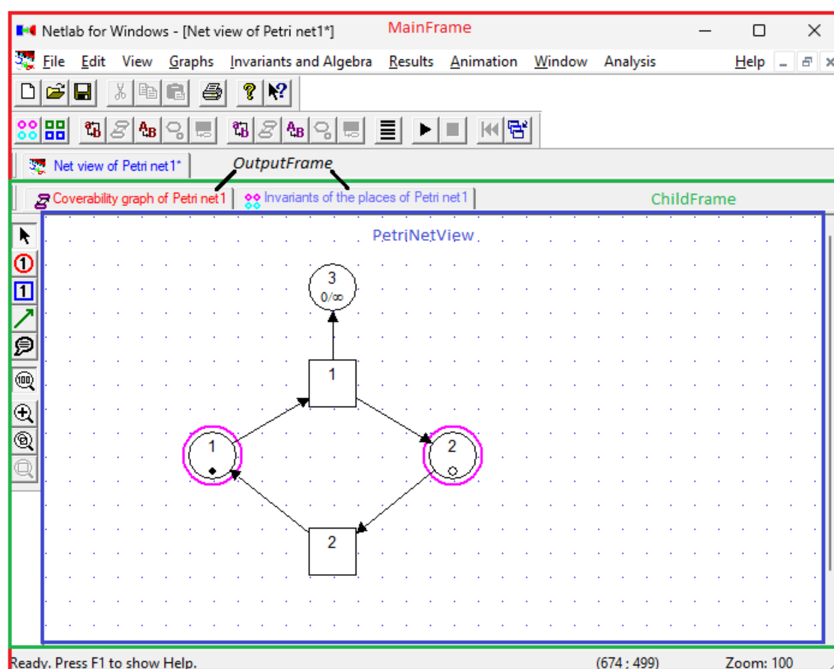
V rámci tejto práce bola medzi analýzy pridaná spätná analýza pre vyšetrenie pokryteľnosti špecifikovaného značenia. Jej implementácia je bližšie popísaná v kapitole 8.

7.3.3 MFC aplikácia

Časť zabezpečujúca interakciu s užívateľom implementovaná pomocou MFC (5) a *document/view* architektúry. Základom je trieda `CNetlabApp`. Dedí priamo z MFC `CWinApp` a jej inštancia je preto definovaná ako globálna premenná. Jej zodpovednosťou je inicializovať ostatné časti aplikácie, pripojiť dynamické knižnice a držať aktuálny stav. Stav okrem iných častí zahŕňa v prvom rade dokumenty Petriho siete s informáciami o rozpracovaných modeloch.

Výzor UI prvkov je definovaný v súbore *Natlab.rc*. V generovanom „Resource Script“ súbore sa nachádzajú dáta o výzore a rozložení jednotlivých prvkov, ale aj číselný identifikátor správy, ktorá je vyvolaná interakciou s ním. Pre editáciu tohto súboru sa takmer výhradne používa grafický editor, ktorý je súčasťou Visual Studia. Obsluha správ je implementovaná v triedach, podľa grafického prvku, ktorý obsluhujú. Ďalej budú uvedené práve tieto triedy.

Po spustení aplikácie sa vždy vytvorí práve jedna inštancia triedy `CMainFrame`. Ide o hlavné okno a celá aplikácia sa predvolene zobrazuje v ňom. Možno pozorovať na obrázku 7.4. Okrem toho sú jej súčasťou panely nástrojov, ktoré okrem iného umožňujú vytvoriť alebo načítať dokument.



Obr. 7.4: Časti grafického rozhrania Netlabu

Pre každý takto spravovaný dokument sa vytvára záložka `CChildFrame` so zobrazením editovanej siete a ďalším panelom nástrojov pre jej editáciu.

Podľa *document/view* architektúry trieda `CPetriNetDoc` definuje dokumenty obsahujúce dáta modelu Petriho siete popísaného v časti 7.3.1 a výsledky vykonaných analýz.

View nad dokumentom popisuje trieda `CPetriNetView`. Implementuje reakcie na primitívne správy ako sú kliknutie myšou, alebo stlačenie klávesy a definuje pre nich obslužné metódy. Tie otvárajú dialógy, menu, upravujú sieť priamo či vyberú časť siete.

Dialógov takto otvoriteľných je tu hneď niekoľko. Pre každý exituje samostatná obslužná trieda:

- `CNodeDialog`, `CEdgeDialog` a `CCommentDialog` pre úpravu vlastností elementov siete.
- `CScaleDialog` a `CRotateDialog` pre zmenu veľkosti a rotáciu vybranej časti siete.
- `CPNMLGraphicsDialog` na úpravu grafických vlastností prvkov.

Okrem toho je tu ešte implementovaných niekoľko ďalších dialógov dostupných z menu.

- `CAboutDialog` zobrazujúci informácie o aplikácii.
- `CGridWidthDialog` pre úpravu rozostupu mriežky.
- `CZoomDialog` pre špecifikovanie úrovne priblíženia na grafický model.
- `CPetriNetPrintDialog` získava extra informácie pre tlač dokumentu.

Analýzy nad vytvoreným modelom sú volané z menu. Spúšťajú sa nad aktuálne vybraným zobrazením, kde klik na analýzu vyvolá správu zachytenú objektom `CChildFrame`, ktorý analýzu spustí a vytvorí `COutputFrame` objekt zobrazujúci výsledky v textovej forme. Vytvorený objekt analýzy uloží v rámci dokumentu.

Kapitola 8

Implementácia spätnej analýzy

V rámci tejto práce bola do aplikácie Netlab pridaná analýza pokryteľnosti značenia za pomoci spätného priechodu grafom. V tejto kapitole je popísaná implementácia UI prvkov pre spustenie analýzy a analýza samotná, spolu so spôsobom testovania.

Princíp tohto algoritmu už bol priblížený v časti 4.3.3 a ako už bolo spomenuté, tak cieľom je zistiť pre špecifikovaný stav jeho pokryteľnosť z počiatočného stavu.

8.1 Spustenie

Analýza sa spustí týmto spôsobom: Najprv sa vyberie sieť určená pre analýzu s inicializovaným počiatočným stavom. Následovne sa v menu klikne na možnosť "Backward Analysis". Otvorí sa dialógové okno, pre nastavenie cieľového stavu a možnosťou zaškrtnúť voľbu pre zahrnutie logov o priebehu v textovej reprezentácii výsledkov. Po potvrdení týchto nastavení sa analýza spustí.

Tlačidlo v menu bolo pridané prostredníctvom grafického editora pre editáciu súboru *Natlab.rc* a definovalo sa tam ID správy pre obsluhu stlačenia tlačidla. Obslužná metóda `OnBackwardAnalysis` bola pridaná do mapy správ v triede `ChildFrame`. Jej vykonanie sa skladá z troch častí:

1. Získanie cieľového stavu prostredníctvom dialógu.
2. Inicializácia analýzy a jej spustenie.
3. Vytvorenie novej záložky s textovým výstupom.

Vytvorenie dialógového okna pre získanie informácií o cieľovom stave sa deje vytvorením objektu triedy `BackwardAnalysisDialog`, nad ktorým sa volá metóda `DoModal`. Táto metóda blokuje postup, dokým nie je dialóg ukončený. Keď sa tak stane, vráti návratovú hodnotu signalizujúcu spôsob ukončenia. Ak tento kód nie je OK, obsluha stlačenia sa ukončí. Toto je situácia, napríklad, keď sa stlačí tlačidlo *Cancel* alebo zavrie dialógové okno krížikom. V opačnom prípade sa pokračuje ďalej inicializáciou analýzy.

Dialógové okno bolo do projektu pridané rovnako ako položka menu cez grafický editor. Okrem okna samotného bol pridaný aj spolu s tabuľkou `CMFCPropertyGridCtrl`. Táto akcia automaticky predgenerovala kód triedy a tá bola doplnená o potrebnú funkcionalitu. Pre inicializáciu sa spravil *„override“* metódy `OnInitDialog`, kde sa pridala inicializácia požadovaných položiek na vyplnenie. Pre uchovanie výsledkov obsahu dialógu aj po jeho

zatvorení sa rozšírila metóda `DoDataExchange` o uloženie týchto prvkov. A pre prístup k nim sa vytvorili príslušné metódy.

Ak skončí metóda `DoModal` dialógového okna potvrdením a vrátením tak návratového kódu OK, tak sa pokračuje vytvorením inštancie triedy `BackwardAnalysis`. V konštruktore berie ukazovateľ na model Petriho siete aby vedela nad akou sieťou analýzu robí a aký je počiatočný stav. Nasleduje volanie jej metódy `Run` s parametrom vyšetrovaného značenia.

8.2 Beh

Algoritmus postupne vyhľadáva sekvenciu prechodov z počiatočného stavu do špecifikovaného cieľového stavu prostredníctvom hrubej sily. Tento proces zahŕňa prehľadávanie stavového priestoru aplikovania možných zmien na aktuálny stav. V kontexte Petriho sieti sa za stav považuje značenie siete, zatiaľ čo za zmenu sa považuje vykonanie prechodu.

8.2.1 Reprezentácia stavu

Pre reprezentáciu stavov sa implementovala trieda `State` držiaca vektor značenia a postupnosti ukazovateľov na prechody aplikovaných na daný stav pre uchovanie výslednej cesty. Trieda ďalej definuje sadu metód nezbytných pre analýzu.

- `IsCoveredBy` metóda berie na vstupu druhý stav a overuje či objekt je ním pokrytý. Porovnáva ich značenia a vracia kladný výsledok ak má vo všetkých miestach rovnaké alebo menšie značenie.
- `CoverAny` berie na vstupu množinu stavov a overuje či objekt pokrýva niektorý z nich.
- `CapacityExceeded` pre overenie kapacity. Berie vektor kapacity ako argument.

Pre ukladanie zoznamu stavov už prehľadaných a pripravených na prehľadanie sa definovala trieda `StateList`. Derivuje `std::list` a pridáva metódu pre odstránenie všetkých stavov pokrývajúce jeden špecifický. Má názov `RemoveIfCovers`. Pri prehľadávaní sa s listom pracuje ako s FIFO frontou, to znamená, že prehľadávanie sa vykonáva do šírky, takže nájdená cesta je vždy optimálna s ohľadom na počet potrebných vykonaní prechodov.

8.2.2 Algoritmus

Jadro algoritmu je možno vidieť na výpise 8.1. Pri inicializácii sa vytvoria dva listy.

- *Open* uchovávajúci zoznam stavov pre spracovanie. Na začiatku sa tam vloží vyšetrovaný stav.
- *Closed* držiaci zoznam stavov už spracovaných. Neobsahuje však všetky prehľadané stavy, ale len ich minimálnu potrebnú podmnožinu. Za ňu sa berie množina vzájomne nepokrytých stavov. V prípade, že by sa tam mali nachádzať dva stavy, kde jeden pokrýva druhý, tak sa ponechá len ten pokrytý.

Algoritmus iteratíve prehľadáva stavový priestor, dokiaľ nie je zoznam Open prázdny, v tom prípade končí s neúspechom. Ak aktuálne spracovávaný stav je pokrytý počiatočným stavom siete, tak je uložená cesta a výsledkom je, že vyšetrovaný stav je pokryteľný.

Pre spracovávaný stav sa vykoná kontrola existencie stavu v množine *Closed*, ktorý je pokrytý spracovávaným. Ak sa taký nájde, tak sa pokračuje ďalšou iteráciou. V opačnom prípade sa volajú metódy pre aktualizáciu listov.

- `UpdateOpenList`, pre rozgenerovanie aktuálneho stavu a pridanie novo vygenerovaných stavov do zoznamu *Open*.
- `UpdateClosedList` pre pridanie stavu do množiny *Closed* a odstránenie z nej stavov, ktoré pokrývajú novo pridaný.

```
StateList openList;
StateList closedList;
openList.emplace_back(targetState);

while(!openList.empty())
{
    State state = openList.front();
    openList.pop_front();

    if (state.IsCoveredBy(initialState))
    {
        mPath = state.Path();
        mIsCoverable = true;
        break;
    }

    if (!state.CoverAny(closedList))
    {
        UpdateOpenList(openList, state);
        UpdateClosedList(closedList, state);
    }
}
```

Výpis 8.1: Jadro implementácie algoritmu spätnej analýzy

8.2.3 Generovanie stavov

Generovanie stavov sa vykoná aplikáciou každého prechodu siete na stav spätne. To znamená, že výstupné podmienky prechodu, ktoré normálne pridávajú značky do siete, z neho odoberú a u vstupných rovnako naopak.

Prechody sa aplikujú všetky bez ohľadu na vstupné podmienky. Takže sa môže stať situácia, kde sa má odobrať viacero značiek než sa tam nachádza. V takom prípade sa ako výsledok nastaví počet značiek v mieste na nula.

Medzi prechodom a miestom sa môžu nachádzať aj vstupné aj výstupné podmienky zároveň. V takom prípade sa najprv aplikujú spätne výstupné podmienky prechodu odberajúce z miest značky a až potom vstupné podmienky pre pridanie. Napríklad, ak by stav mal dve značky v mieste a prechod by odoberal tri a pridával tri, tak sa najprv odoberú tri. Miesto má nula značiek. Nasledovne sa tri pridajú. Takže sa vytvoril stav, kde je prechod možné vykonať, ale jeho vykonaním sa dostane stav, kde je počet značiek vyšší ale pokrývajúci pôvodný stav.

Implementácia podporuje špecifikovanie kapacity. Preto sa vždy po spätnej aplikácií prechodu spraví kontrola pomocou už spomenutej metódy `CapacityExceeded` a ak by malo byť nejaké miesto preplnené, tak sa takýto stav ihneď zahadzuje.

8.3 Reprezentácia výsledkov

Po ukončení behu algoritmu sa volá metóda `ChildFrame` triedy `CreateTextOutput`, ktorá volá metódy rozhrania `OutputBase`, pre vytvorenie textového výstupu vo forme okna rovnako ako pri ostatných analýzach. Výsledky obsahujú informácie o vstupných parametroch, informáciu či bola nájdená cesta od počiatku až do cieľa a ak áno, tak je vypísaná postupnosť aplikovania prechodov od počiatočného značenia až po stav ktorý pokrýva vyšetrované značenie.

Pri spustenom logovaní sa navyše vypíše stav *open* a *closed* setu po každom kroku.

8.4 Testovanie

Pre overenie správnosti fungovania algoritmu bola vytvorená sada testov v novo vytvorenom VS projekte *Netlab.Tests*.

Na spúšťanie testov sa tu používajú parametrizované testy *GoogleTest* frameworku. Špecificky makrá `TEST_P` pre deklarovanie parametrizovaného testu a pre ich vytvorenenie zo zoznamu parametrov sa použije makro `INSTANTIATE_TEST_CASE_P`.

Parameter je typu `std::string` a obsahuje cestu k súboru bez prípony. Pretože ako parameter sa berie trojica súborov, ktorých názov sa líši iba príponou. Všeobecne majú ich názvy tvar: *path_to_file.(in/out/net)*. Súbor s príponou *.net* je súbor s uloženým modelom Petriho siete vo formáte *PNML*, ktorý je možno získať exportom priamo z nástroja Netlabu. *.in* súbor obsahuje vektor značenia vyšetrovaného stavu a súbor *.out* pozostáva z textového výstupu, ktorý je získateľný z algoritmu. Predstavuje očakávaný výsledok. Test samotný berie jeden parameter s názvom testu a jeho vykonanie sa skladá z troch častí:

1. Načítanie modelu siete a cieľového miesta zo súborov.
2. Spustenie spätnej analýzy s načítanými vstupmi.
3. Porovnanie výsledku analýzy s očakávaným výstupom načítaného zo súboru.

Zoznam testov sa vytvára prehľadaním zadaného adresára. Skontrolujú sa názvy všetkých súborov. Za test sa považuje trojica súborov popísaných vyššie a uloží sa cesta. Ak sa tam aspoň jeden z nich nenachádza, tak sú ostatné ignorované.

Model siete pre test je získaný použitím xml knižnice na prečítanie súboru s príponou *.net* a triedy modelu `PetriNet`, ktorej sa vytvorí inštancia a z načítaného súboru zoberie štruktúru a značenie siete. Tieto súbory sa vytvárajú exportom modelu priamo z prostredia aplikácie a spôsob ich načítania je pre použitie import funkcionality.

Načítanie súboru *.in* s vektorom cieľového značenia sa vykonáva za pomoci štandardnej funkcionality jazyka *C++*, keďže formát je v celku jednoduchý. To znamená postupnosť celých čísel na jednom riadku oddelených čiarkou. Súbor s očakávaným výstupom *.out* je načítaný obdobným spôsobom.

Tieto metódy pre načítanie zo súborov a vytváranie zoznamu testov boli zámerne oddelené do samostatnej statickej triedy, aby bolo možné funkcionalitu použiť aj pre ďalšie analýzy.

Kapitola 9

Záver

Hlavnými cieľmi tejto práce bolo umožniť ďalší vývoj aplikácie Netlab a pridať implementáciu spätnej analýzy. Pre uľahčenie ďalšieho vývoja bolo vykonaných množstvo menších úprav zdrojových súborov a boli vysvetlene teoretické základy problematiky, ktorú program rieši.

V prvom rade bol vykonaný prieskum zdrojových kódov samotných, pretože dokumentácia bola len obmedzená a to vo forme generovanej dokumentácie priamo z kódu. Z tohto prieskumu vyšlo hneď niekoľko problémov, ktoré bolo potrebné vyriešiť pred ďalšími zmenami. Bola vykonaná migrácia na novšie Visual Studio a nový štandard jazyka C++. Závislosti a štruktúra riešenia bola prerobená, aby sa okrem iného úplne odstránila závislosť na Matlab a minimalizovala závislosť na jazyk Perl. Zdrojové kódy boli prečistené s ohľadom na čitateľnosť. Upravená štruktúra a architektúra aplikácie, je popísaná v rámci tohto textu.

Ako pridaná hodnota pre užívateľa aplikácie bolo implementované rozšírenie aplikácie o spätnú analýzu pre kontrolu pokryteľnosti špecifikovaného značenia modelu Petriho siete. Spôsob spustenia a reprezentácie výsledkov analýzy bol spravený obdobne, ako už u ostatných implementovaných analýz pre konzistentnosť. Na rozdiel od nich však spätná analýza berie okrem modelu samotného aj cieľový stav. Preto rozšírenie zahŕňa aj pridanie dialógu pre získanie stavu od užívateľa.

Tieto informácie o štruktúre a fungovaní aplikácie spoločne s popisom pridania spätnej analýzy môže byť použité ako návod pre ďalšie rozšírenia. Preštudovaním zdrojových kódov ako aj aplikácie samotnej sa ukázalo, že sa tu nachádzajú implementované časti s cieľom podporovať ďalšie rozšírenia Petriho sieti. Špecificky sa jedná o inhibítor hrany alebo modelovanie času. Treba ale brať ohľad na fakt, že súčasné implementácie analýz tieto koncepty nemusia podporovať.

Literatúra

[1] Aachen, R. *Petrinet-Tool Netlab* [online]. 2023 [cit. 20.3.2023]. Dostupné z: https://web.archive.org/web/20210621105511/https://www.irt.rwth-aachen.de/cms/IRT/Studium/Downloads/~osru/Petrinetz-Tool-Netlab.

[2] Baier, C. a Katoen, J.-P. *Principles of Model Checking*. Január 2008. ISBN 978-0-262-02649-9.

[3] Bingham, J. A New Approach to Upward-Closed Set Backward Reachability Analysis. *Electronic Notes in Theoretical Computer Science*. 2005, zv. 138, č. 3, s. 37–48. DOI: https://doi.org/10.1016/j.entcs.2005.01.045. ISSN 1571-0661. Proceedings of the 6th International Workshop on Verification of Infinite-State Systems (INFINITY 2004). Dostupné z: https://www.sciencedirect.com/science/article/pii/S1571066105051996.

[4] Bobbio, A. a C, K. System Modelling With Petri Nets. December 2002. DOI: 10.1007/978-94-009-0649-5_6.

[5] Busi, N. Analysis issues in Petri nets with inhibitor arcs. *Theoretical Computer Science*. 2002, zv. 275, č. 1, s. 127–177. DOI: https://doi.org/10.1016/S0304-3975(01)00127-X. ISSN 0304-3975. Dostupné z: https://www.sciencedirect.com/science/article/pii/S030439750100127X.

[6] Czerwiński, W., Lasota, S., Lazić, R., Leroux, J. a Mazowiecki, F. The Reachability Problem for Petri Nets Is Not Elementary. *Journal of the ACM (JACM)*. 2021, zv. 68, č. 1, s. 7. DOI: 10.1145/3422822. HAL Id: hal-03436240.

[7] Esparza, J. Decidability and complexity of Petri net problems - An introduction. *LNCS*. September 2000, zv. 1491. DOI: 10.1007/3-540-65306-6_20.

[8] Jacobsen, L., Jacobsen, M., Møller, M. a Srba, J. Verification of Timed-Arc Petri Nets. In:. Január 2011, s. 46–72. DOI: 10.1007/978-3-642-18381-2_4. ISBN 978-3-642-18380-5.

[9] Lukács, G. a Bartha, T. Formal Modeling and Verification of the Functionality of Electronic Urban Railway Control Systems Through a Case Study. *Urban Rail Transit*. Dec 2022, zv. 8, č. 3, s. 217–245. DOI: 10.1007/s40864-022-00177-8. ISSN 2199-6679. Dostupné z: https://doi.org/10.1007/s40864-022-00177-8.

[10] 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. DOI: 10.1109/TCOM.1976.1093424.

[11] Michael, J., Drusinsky, D. a Wijesekera, D. Formal Verification of Cyberphysical Systems. *Computer*. Los Alamitos, CA, USA: IEEE Computer Society. sep 2021, zv. 54, č. 09, s. 15–24. DOI: 10.1109/MC.2021.3055883. ISSN 1558-0814.

[12] Microsoft. *MFC application architecture classes* [online]. 2023 [cit. 15.3.2023]. Dostupné z: http://web.archive.org/web/20230415085902/https://learn.microsoft.com/en-us/cpp/mfc/mfc-application-architecture-classes?view=msvc-170.

[13] Microsoft. *MFC Desktop Applications* [online]. 2023 [cit. 15.3.2023]. Dostupné z: https://web.archive.org/web/20230315194400/https://learn.microsoft.com/en-us/cpp/mfc/mfc-desktop-applications?view=msvc-170.

[14] Murata, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*. 1989, zv. 77, č. 4, s. 541–580. DOI: 10.1109/5.24143.

[15] Peterson, J. L. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[16] Petri, C. A. *Kommunikation mit Automaten*. Germany, 1962. Dizertačná práca. University of Bonn.

[17] Prosise, J. *Programming Windows with MFC*. Microsoft Press, 1999. Microsoft programming series. ISBN 9781572316959. Dostupné z: https://books.google.cz/books?id=lSePAQAACAAJ.

[18] Ramamoorthy, C. a Ho, G. Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets. *IEEE Transactions on Software Engineering*. 1980, SE-6, č. 5, s. 440–449. DOI: 10.1109/TSE.1980.230492.

[19] Reisig, W. *Understanding Petri nets. Modeling techniques, analysis methods, case studies*. Júl 2013. ISBN 978-3-642-33277-7.

[20] Weber, M. a Kindler, E. The Petri Net Markup Language. In: Ehrig, H., Reisig, W., Rozenberg, G. a Weber, H., ed. *Petri Net Technology for Communication-Based Systems: Advances in Petri Nets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, s. 124–144. DOI: 10.1007/978-3-540-40022-6_7. ISBN 978-3-540-40022-6. Dostupné z: https://doi.org/10.1007/978-3-540-40022-6{_}7.

Príloha A

Príprava prostredia a kompilácia

V tejto prílohe je popísaný postup pre preloženie zdrojových súborov. Sú tu uvedené verzie nástrojov, na ktorých bol tento postup testovaný, čo nevylučuje možnosti kompilácie s novšími alebo staršími verziami.

A.1 Prerekvizity

Visual Studio 2022 Community (17.4.3). Toto vývojové prostredie umožňuje po inštalácii otvoriť súbor riešenia s príponou *.sln*. Je vyžadovaný "Desktop developement with C++"*workload* a príslušný C++ MFC(v143) balíček. Po otvorení sa v prípade, že chýbajú nejaké balíčky, zobrazí upozornenie a po kliknutí na neho možnosť ich automaticky inštalovať. Ak by tomu tak nebolo, tak je potrebné ich nainštalovať manuálne pomocou VS inštalátora. Po ukončení inštalácie a znovu načítaniu riešenia by ho malo byť možné otvoriť a kompilovať projekty. Výnimkou je projekt *Help*, pre generovanie pomocníka, ktorý vyžaduje na zariadení mať ešte inštalovaný Perl (v5.32.1). Toto však nie je nutné, pretože na projekte *Help* žiadny iný nezávisí a vývoj aj kompilácia je možná aj bez neho.

A.2 Kompilácia a spustenie

Po splnení všetkých prerekvizít by malo byť možné zostaviť celé riešenie a spustiť aplikáciu za predpokladu, že projekt *NetlabApp* je nastavený ako štartovací projekt. Všetky potrebné preložené súbory sa nachádzajú v adresári *bin*.

A.3 Testy

Projekt s testami *Netlab.Tests* je závislý na Google testoch inštalovaný ako nugget. Je možné ich spustiť buď priamo cez Visual studio alebo alternatívne prekladom tohto projektu sa vytvorí *Netlab.Tests.exe* pre ich spustenie.