ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Sadloň** Jméno: **Richard** Osobní číslo: **453455**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Software pro automatické generování modelů systémů Internet of Things (IoT)

Název diplomové práce anglicky:

Software For the Automated Generation of Models of Internet of Things (IoT) Systems

Pokyny pro vypracování:

Navrhněte a implementujte software pro automatické generování modelů inspirovaných reálnými systémy Internet of Things (IoT). Software bude sloužit pro ověření experimentálních algoritmů generujících testovací scénáře pro ověření systémů IoT v případě omezeného nebo nedostupného síťového připojení.
Pro vývoj softwaru analyzujte existující systémy a navrhněte vstupy, kterými bude možné ovlivnit vlastnosti výsledného modelu. Zaměřte se na modely reprezentujících procesy a stavy popisovaných systémů.
Umožněte v softwaru možnost nejen vygenerovat zcela nový model, ale také přizpůsobit model již existující, aby odpovídal zadaným vlastnostem.
Strukturu generovaného modelu implementujte jako prostý orientovaný graf, který může obsahovat cykly. U možností formátu vstupního a výstupního souboru s modelem systému umožněte podporu platformy Oxygen.

Seznam doporučené literatury:

1. Madakam, S., Lake, V., Lake, V., & Lake, V. Internet of Things (IoT): A literature review. Journal of Computer and Communications, 3(05), 2015.
2. Tsiatsis, Vlasios, et al. Internet of Things : Technologies and Applications for a New Age of Intelligence. London: Academic Press, 2019.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Matěj Klíma, laboratoř inteligentního testování systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **10.02.2021** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **30.09.2022**

Ing. Matěj Klíma	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce		podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Diplomová práca

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická

Software pro automatické generování modelů systémů Internet of Things (IoT)

Bc. Richard Sadloň

Školiteľ: Ing. Matěj Klíma
August 2021

Poďakovanie

Ďakujem školiteľovi Ing. Matějovi Klímovi za pravidelné konzultácie a rady pri vypracovaní tejto práce. Taktiež by som rád poďakoval Ing. Václavovi Rechtbergerovi za dodanie užitočných informácii pri návrhu stavového modelu.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval samostatne a že som uviedol všetky použité informačné zdroje v súlade s Metodickým pokynom č. 1/20 o dodržiavaní etických princípov pri príprave vysokoškolských záverečných prác.

V Prahe, 12. 8. 2021

Abstrakt

Cieľom tejto práce bolo navrhnúť a implementovať software pre generovanie alebo rozšírenie grafov, ktoré sú inšpirované reálnymi IoT systémami. Hlavným využitím má byť testovanie experimentálnych algoritmov vyvíjaných vrámci skupiny STILL na katedre počítačov FEL ČVUT. Tieto algoritmy sa zameriavajú na testovanie v prípade výpadku alebo obmedzenia sieťového pripojenia.

Práca v úvode obsahuje vymedzenie základných pojmov z teórie grafov, stručný popis IoT systémov, popis problému testovania v prípade výpadku alebo obmedzeného sieťového pripojenia, rešerš IoT systémov zakončenú zhodnotením požiadavok na výslednú aplikáciu. Ďalej nasleduje popis navrhnutého riešenia a jeho realizácie.

Implementácia je v programovacom jazyku Java. Generované alebo rozširované grafy môžu obsahovať cykly, v prípade rozšírenia existujúceho grafu je tento problém je riešený pomocou silne súvislých komponent a ich detekciou využitím Tarjanového algoritmu. K overeniu, že výsledný graf obsahuje presne chcený počet cyklov bol použitý Johnsnov algoritmus na detekciu všetkých cyklov v orientovanom grafe.

Kľúčové slová: Internet vecí, generátor grafov, Tarjanov algoritmus, Oxygen, testovanie na základe modelu.

Školiteľ: Ing. Matěj Klíma
laboratoř inteligentního testování systémů FEL

Abstract

The aim of this work was to design and implement software for generating or extending graphs that are inspired by real IoT systems. The use of this product can be the testing of experimental algorithms developed within the STILL group at the Department of Computers, FEE CTU. These algorithms focus on testing in the event of a network connection failure or limitation.

The work in the introduction contains the definition of basic concepts of graph theory, a brief description of IoT systems, a description of the problem of testing in case of failure or limited network connection, a search of IoT systems ended by evaluating the requirements for the resulting application. Next, the description of the proposed solution and its implementation.

The implementation is in the Java programming language. Generated or extended graphs can be cyclic, in case of extended graph this problem is solved using strongly connected components and their detection using the Tarjan's algorithm. To verify that the resulting graph contains the exact desired number of cycles, a Johnson's algorithm was used to detect all cycles in the oriented graph.

Keywords: Internet of Things, graph generator, Tarjan's algorithm, Oxygen, model-based testing.

Title translation: Software For the Automated Generation of Models of Internet of Things (IoT) Systems

Obsah

Obrázky

Tabuľky

Kapitola 1
Úvod

Neoddeliteľnou súčasťou vývoja software je fáza testovania. Pri testovaní môžme namodelovať vyvíjaný software, tento model nazývame SUT[1] a generujeme z neho kombinácie v ktorých sa systém môže ocitnúť. Tieto kombinácie sú dôležitým podkladom pre testovacie scenáre. Môžme ísť napríklad cestou všetkých možných kombinácii uzlov a hrán v modeli. Tento prístup je avšak v praxi častokrát neaplikovateľný, kvôli časovej a finančnej náročnosti. Reálne systémy môžu byť príliš veľké. Preto sa pri testovaní snažíme výslednú množinu kombinácii zmenšiť.

Rovnako ako môžme modelovať software môžme modelovať aj IoT systém a takisto vytvárať tieto kombinácie. Projekt Oxygen vytvorený a naďalej inovovaný katedrou počítačov na ČVUT FEL obsahuje experimentálne algoritmy, ktoré generujú testovacie scenáre IoT systému s ohľadom na zóny v ktorých môže dôjsť k obmedzeniu alebo výpadku sieťového pripojenia. Tieto algoritmy je potrebné overiť na "reálnych"modeloch, ideálne na čo najväčšom počte rôznych modelov a taktiež aj na modeloch, ktoré sú rozsiahlejšie. Okrem overenia môže byť do budúcna taktiež potreba porovnávať algoritmy medzi sebou, napríklad z hľadiska času. Nutnou podmienkou pre overenie algoritmov je existencia modelov vo vhodnom formáte. Projekt Oxygen umožňuje tieto modely ručne nakresliť a uložiť tak, aby si s nimi algoritmy poradili. Cieľom generovania testovacích scenárov je automatizácia testovania, manuálne vytváranie modelov je potrebné nahradiť automatickým generovaním. To bude do budúcna šetriť čas a zníži chybovosť pri vytváraní modelov.

Cieľom tejto diplomovej práce je vyriešiť problém tvorenia modelov pre testovanie vyššie spomenutých algoritmov. Výsledkom práce je Java desktopová aplikácia, ktorá umožní vygenerovať model podľa zadaného vstupu a taktiež rozšíriť model už existujúci o požadované vlastnosti.

Vzhľadom nato, že model reprezentujeme grafom so špecifickými vlastnosťami teoretická časť textu začína definovaním používaných pojmov z teórie grafov, nasleduje stručným zoznámením so svetom Internetu vecí(IoT), rešeršou IoT, popisom Oxygenu a problému testovania IoT v prípade obmedzeného alebo nedostupného sieťového pripojenia. Nasleduje praktická časť ktorá obsahuje špecifikáciu požiadavok na vyvíjaný software, popis architektúry a použité technológie, popis implementácie vrátane rozboru vytvorených algo-

[1] akronym z anglického: System under test

ritmov. Text končí kapitolou venovanou testovaniu vyvinutého produktu a záverečným zhodnotením.

Kapitola 2
Popis problému

V tejto kapitole je vysvetlenie niektorých pojmov z teórie grafov, potrebných v zvyšku textu. Nasleduje popis Internetu vecí, vysvetlenie problematiky testovania IoT systému v prípade obmedzeného alebo nedostupného sieťového pripojenia vrátane popisu potrebného modelu. Kapitola končí rešeršou IoT systémov s hlavným cieľom popis spomínaného modelu viac špecifikovať.

2.1 Grafy

Vzhľadom na nasledujúci text je potrebné definovať niektoré pojmy z teórie grafov. V tejto sekcii sa opierame o informácie z [1].

Graf je trojica (V, E, ε), kde V je neprázdna konečná množina uzlov, E je konečná množina hrán a ε je zobrazenie, ktoré v prípade, že sa jedná o orientovaný graf tak každej hrane $e \in E$ priraďuje usporiadanú dvojicu uzlov, v prípade, že sa jedná o neorientovaný graf tak každej hrane $e \in E$ priraďuje množinu dvoch uzlov. Zobrazenie ε sa nazýva vzťah incidencie. V prípade orientovaného grafu prvý uzol z usporiadanej dvojice nazývame ako počiatočný uzol hrany a druhý uzol ako koncový uzol hrany.Ak hrana vedie sama do seba tak hovorime o smyčke. Ak hranám v grafe priradíme ohodnotenie tak hovorime o ohodnotených hranách. V zvyšku sekcie budeme pod pojmom graf myslieť trojicu (V, E, ε) ako je definovaná vyššie. Uzly v_i budú z množiny V a hrany e_i z množiny E.

Paralelné hrany - ak v grafe sú dve rôzne hrany pre ktoré sa rovnajú vzťahy incidencie, potom su hrany paralelné.

Prostý graf je graf, ktorý neobsahuje paralelné hrany.

Orientovaný sled je postupnosť uzlov a hrán $(v_1, e_1, v_2, ..., v_{k-1}, e_k, v_k)$ v ktorej pre všetky $i = 1, 2, ..., k-1$ platí, že v_i je počiatočný uzol hrany e_i a v_{i+1} je koncový uzol hrany e_i, v prípade **neorientovaného sledu** sú v_i a v_{i+1} krajné uzly hrany e_i. Ak počet uzlov je väčší ako 1 a $v_1 = v_k$ potom hovoríme o **uzatvorenom slede**. Uzol v_1 považujeme za počiatočný uzol sledu a v_k za koncový uzol sledu.

Ťah je sled, v ktorom sa neopakujú hrany. V závislosti od toho či je sled orientovaný alebo neorientovaný delíme ťah na orientovaný a neorientovaný.

Cesta je ťah, v ktorom sa neopakujú uzly. Môže byť orientovaná alebo neorientovaná. Vzhľadom nato, že cesta je ťahom a tým pádom aj sledom, môžme počiatočný a koncový uzol sledu považovať za počiatočný a koncový uzol cesty. Platí, že v ceste sa môže počiatočný a koncový uzol rovnať potom hovoríme o uzatvorenej ceste. Uzatvorená orientovaná cesta je **cyklus**.

Neorientovaný graf je **súvislý** ak existuje cesta z uzlu v_1 do uzlu v_2 pre každé dva uzly v_1 a v_2 v grafe. Orientovaný graf je **súvislý** ak jeho prevedenie na neorientovaný graf, teda zabudnutie na smer hrán je graf súvislý. Povieme, že orientovaný graf je **silne súvislý** ak pre každé dva uzly v_1 a v_2 existuje orientovaná cesta z v_1 do v_2.

Podgraf grafu dostaneme tak, že z pôvodného grafu vynecháme niektoré alebo žiadne uzly a niektoré alebo žiadne hrany s tou podmienkou, že ak necháme hranu e tak necháme aj krajné uzly tejto hrany. Majme daný orientovaný graf G, množinu uzlov B nazveme ako **komponentu silnej súvislosti** ak je to maximálna množina uzlov taká, že podgraf indukovaný B je silne súvislý. Majme daný orientovaný graf G, povieme, že uzol v je **koreň** grafu G, ak pre každý uzol v_i z V existuje orientovaná cesta z v do v_i.

2.2 Internet vecí

Cieľom tejto sekcie je zoznámiť čitateľa s pojmom Internet vecí, ukázať najpoužívanejšie technológie, možností využitia a dôvody na zaistenie kvality.

2.2.1 Popis

Pojem Internet vecí ako prvý použil Kevin Ashton v roku 1999. Učinil tak pri svojej prezentácii pre spoločnosť Procter & Gamble, kde popisoval ako zavedenie RFID tagov môže zlepšiť chod spoločnosti. Ďalej v roku 2009 napísal článok [2], v ktorom vysvetľuje ako počítače a tým pádom aj internet boli závislé na údajoch od užívateľov(ľudí). Zjednodušene povedané informácie, ktoré sme mali boli vytvorené ľudmi, napríklad tým, že ich napísali, odfotili a podobne. Myšlienkou bolo, že keď pripojíme k internetu zariadenia(veci) budeme mať presnejšie informácie a budeme ich mať omnoho viac. Presnejšie informácie pretože ľudia môžu robiť chyby, viac informácií pretože ľudia sú časovo limitovaný ostatnými povinnosťami. Ďalej v texte budeme označovať Internet vecí zaužívaným akronymom z anglického názvu[1] a to IoT. Podľa [3] stále nemáme obecnú definíciu pojmu IoT, ďalej sa budeme držať nasledujúcej definície:

[1] Internet of Things

"Otvorená a komplexná sieť inteligentných objektov, ktoré sú schopné automatickej organizácie, zdieľania informácií, údajov a zdrojov, reagovať a konať podľa situácie a zmien v prostredí."
(preložená priama citácia z [3])

Pojem Internet vecí je zložení z dvoch slov, prvým je Internet čo je globálny systém vzájomne prepojených počítačových sietí, ktoré používajú štandardnú internetovú sadu protokolov (TCP / IP), ktorá slúži miliardám používateľov na celom svete. Je to sieť sietí, ktorá sa skladá z miliónov súkromných, verejných, akademických, obchodných a vládnych sietí miestneho až globálneho rozsahu, ktoré sú prepojené a široká škála elektronických, bezdrôtových a optických sieťových technológií [4]. Zatiaľ čo vecou môžme označiť čokoľvek živé ako napríklad človek, zviera alebo aj rastlinu. Neživé ako žiarovka, chladnička, a podobne. Nato aby sme tieto objekty mohli nazvať vecami v zmysle IoT potrebujeme aby boli jasne identifikovateľné, schopné zbierať údaje a komunikovať s ostatnými zariadeniami/internetom to docielime pripojením vhodného zariadenia [5]. Môžu síce pracovať aj bez pripojenia k internetu, no pre plnohodnotné využívanie potenciálu, je pripojenie k internetu a prepojenie s ďalšími zariadeniami nevyhnutné [5].

Rast IoT systémov je dnes jasne viditeľný, napríklad *Heslop* uvádza odhad 125 miliard zariadení do roku 2030 [7]. Pre porovnanie podľa spoločnosti *Cisco* ich počet bol 12.5 miliardy v roku 2010 [8].

2.2.2 Technológie

Definícia IoT hovorí o inteligentných objektoch a o tom čoho by mali byť schopné, text vyššie avšak tvrdí, že vo svete IoT môže byť vecou(objektom) chápané v podstate čokoľvek. Cestou ako z obyčajnej veci urobiť inteligentný objekt je pridanie vhodného zariadenia. Jednotlivé zariadenia musí ísť jasne identifikovať, k tomu slúžia označovacie technológie ako napríklad RFID [3]. K zberu a generovaniu dát slúžia senzory a/alebo aktuátory. Nato aby zariadenia boli schopné komunikovať potrebujú sieť. Sieť pozostáva z uzlov a spojníc medzi nimi pomocou ktorých si uzly vymieňajú informácie. Zariadenia môžme považovať za uzly. Nato aby si uzly v sieti mohli vymieňať informácie musí ich byť možné jednoznačne adresovať nato slúžia IP adresy. Sieťové technológie vieme rozdeliť na drôtové a bezdrôtové, príkladom drôtových je ethernet, medzi bezdrôtové patrí napríklad wi-fi, celulárne siete, bluetooth alebo zigbee. Dáta zo zariadení sa podľa potreby ďalej môžu napríklad ukladať do databázy, validovať, spracovávať alebo analyzovať [5]. Vyššie spomenuté technológie sú podrobnejšie popísane v nasledujúcom texte.

RFID

Rádiofrekvenčná identifikácia (RFID) je systém, ktorý prenáša identitu objektu pomocou rádiových vĺn vo forme sériového čísla. Medzi hlavné komponenty patrí tag, anténa a čítačka. Tag a anténa sú umiestnené na cieľovom

zariadení vo forme malého štítku. Tag slúži ako jednoznačný identifikátor, anténa slúži na komunikáciu s čítačkou. Hlavným využitím je jednoznačná identifikácia produktu [5].

Senzory a aktuátory

Senzor je zariadenie, ktoré sleduje vlastností fyzickej entity a prevádza ich na informácie. Aktuátor je zariadenia, ktoré môže meniť vlastností fyzickej entity v reakcii na prijaté informácie [5].

IP

Internetový protocol(IP) je hlavný sieťový protokol používaný na internete. Používané sú dve verzie IPv4 a IPv6. Ip adresy pomáhajú identifikovať zariadenie v sieti. IPv4 protokol poskytuje okolo 4.3 miliardy IPv4 adries. Ipv6 ponúka až 2^{128} adries [3].

Wi-Fi

Wi-Fi[2] je sieťová technológia, ktorá umožňuje počítačom a iným zariadeniam bezdrôtovú komunikáciu [3]. Bez ohľadu nato, že spotrebúva viac energie ako napríklad BLE pri IoT má stále svoje zastúpenie a to hlavne kvôli vysokej prenosovej rýchlosti [5].

Celulárna rádiová sieť

Celulárna rádiová sieť je rádiová telekomunikačná sieť, v ktorej komunikáciu v rozľahlejšej geografickej oblasti zabezpečuje množstvo základňových staníc, ktoré svojím dosahom vytvárajú sústavu vzájomne sa prekrývajúcich buniek. Medzi celulárne siete patria známe technológie ako 2G, 3G, 4G LTE a dnes naberajúce na popularite 5G. Medzi hlavné výhody patrí široké pokrytie a veľký dosah [5].

Bluetooth

Bluetooth je typ bezdrôtovej komunikácie, ktorá sa používa na prenos údajov vysokou rýchlosťou pomocou rádiových vĺn. Nevýhodou tohto protokolu je krátky dosah. Typická rýchlosť prenosu dát je okolo 2 megabitov za sekundu. V priemyselnom IoT musia stroje často odosielať krátke série dát v extrémne hlučných prostrediach. S potenciálne stovkami senzorov a zariadení, ktoré odosielajú dáta, predstavuje Wi-Fi príliš veľa problémov s nastavením. Pre využitie vo svete IoT sa zvlášť hodí jeho nizkoenergetická verzia BLE[3] [10].

[2] Wireless Fidelity
[3] Bluetooth Low Energy

Zigbee

ZigBee je bezdrôtová komunikačná technológia. Jej sieť môže obsahovať až 65 000 zariadení, čo je dvakrát viac, ako podporuje BLE. Zigbee sa primárne používa na domáce automatizačné aplikácie, ako je inteligentné osvetlenie, inteligentné termostaty a monitorovanie domácej energie. Bežne sa používa aj v priemyselnej automatizácii, inteligentných meračoch a bezpečnostných systémoch [9].

2.2.3 Využitie

Táto podsekcia ukazuje možné využitie IoT, opiera sa o informácie z [5] a [11].

- **Riadenie aktív**[4]
  Pri aktívach môžme chcieť vedieť napríklad ich lokáciu alebo ako často sa využívajú [5]. Reálnym príkladom môže byť kontrola tovaru v sklade, vďaka IoT vieme mať informácie, kde a v akom množstve sa tovar nachádza.

- **Priemyslová automatizácia**[5]
  Priemyselný internet vecí je použitie IoT na zlepšenie výroby a kvality výrobkov a častí výrobkov organizácie. Používa sa predovšetkým na informácie o stave strojov a príčinách výskytu chybných častí [5].

- **Inteligentná sieť**[6]
  Inteligentná sieť hovorí o monitorovaní a spravovaní napríklad osvetlenia, dopravných značiek, dopravných zápch, parkovacích miest [5].

- **Inteligentné zdravie**[7]
  Jedným z ďalších využití IoT, je využitie pri zdravotnej starostlivosti. Podľa [11] je najvýznamnejšie využitie v diaľkovom monitorovaní pacienta. Zariadenia IoT môžu automaticky zhromažďovať metriky zdravia, ako sú srdcová frekvencia, krvný tlak, teplota a ďalšie, od pacientov, ktorí sa fyzicky nenachádzajú v zdravotníckom zariadení.

2.2.4 Zaistenie kvality v prostredí IoT

Z predošlej sekcie je zrejmé, že IoT systémy musia spoľahlivo a správne fungovať. Napríklad pri využití vo výrobnej fabrike by nesprávne fungujúci systém mohol spôsobiť vysoké finančné škody, rovnako ak nie viac závažné by mohli byť zlyhania pri monitorovaní zdravia pacientov. Podľa [12] atribúty kvality systému pomáhajú pri plánovaní testov a/alebo tvorbe testovacej stratégie. Z pohľadu tejto práce ukazujú hlavne čo a prečo testovať. Zdroj [13] uvádza nasledujúce atribúty kvality, ktoré sú dôležité v prostredí IoT:

[4] Asset Managment
[5] Industrial Automation
[6] Smart Grid
[7] Smart Health

- **Kompatibilita**
  Miera, do akej si výrobok, systém alebo jeho komponenta môže vymieňať informácie s inými systémami alebo komponentami.
- **Dôvernosť**
  Rozsah, v akom systém zaisťuje, že k údajom majú prístup iba autorizované osoby.
- **Efektívnosť**
  Hovorí o tom, koľko zdrojov bolo potrebných k dosiahnutiu užívateľských cieľov.
- **Inštalovateľnosť**
  Miera efektívnosti, s akou je možné produkt alebo systém úspešne nainštalovať a/alebo odinštalovať v konkrétnom prostredí.
- **Interoperabilita**
  Miera, do akej si môžu dva alebo viac systémov alebo komponentov vymieňať informácie a využívať vymenené informácie.
- **Spoľahlivosť**
  Miera, do akej systém, vykonáva zadané funkcie za stanovených podmienok po stanovenú dobu.
- **Využitie zdrojov**
  Miera, do akej množstvo a typ zdrojov, ktoré systém používa pri vykonávaní svojich funkcií, spĺňa požiadavky.
- **Spokojnosť**
  Miera, do akej systém uspokojuje potreby používateľov.
- **Bezpečnosť**
  Miera, do akej sú informácie a údaje o produkte alebo systéme chránené, takže osobám, iným produktom alebo systémom je poskytovaná správna úroveň prístupu zodpovedajúca ich úrovni autorizácie.
- **Časové správanie**
  Miera odozvy a doby spracovania systému počas vykonávania jeho funkcií.
- **Použiteľnosť**
  Rozsah, v akom môže byť systém používaný na efektívne uspokojivé dosiahnutie stanovených užívateľských cieľov.

2.3 Testovanie IoT a sieťové pripojenie

Táto sekcia zoznamuje s modelovaním IoT systému vzhľadom na možné obmedzenie alebo výpadok sieťového pripojenia. Ako je uvedené v predošlej sekcii tak jednotlivé zariadenia tvoria sieť, pomocou ktorej medzi sebou komunikujú. Podľa [14] je potrebné pri testovaní pokryť nasledujúce situácie:

1. Spojenie komunikácie medzi jednotlivými časťami systému sa preruší.
2. Prerušené spojenie sa opäť obnoví.

Testy by mali skontrolovať, že testovaný IoT systém bude bez ohľadu na vyššie spomenuté situácie fungovať správne. Vhodnou možnosťou je testovanie modelu procesov. Výhodou tohto prístupu je opakované generovanie testovacích scenárov [15]. Model vytvoríme tak, že najprv detekujeme jednotlivé subsystémy v systéme, potom detekujeme business procesy v nich a to ako medzi sebou komunikujú. Model je graf v ktorom jednotlivé procesy tvoria uzly a spojnice komunikácie hrany.

Procesný model IoT systému

Práca [14] vzhľadom na fungovanie algoritmov, ktoré generujú testovacie scenáre z modelu upresňuje ako má model vyzerať.

Model je orientovaný graf označme ho ako G s nasledujúcimi vlastnosťami:

1. G je súvislý.
2. G neobsahuje paralelné hrany ani smyčky.
3. G obsahuje presne jeden koreň.
4. G obsahuje aspoň jeden uzol z ktorého nevedú žiadne hrany.

Koreň grafu budeme ďalej označovať ako **štartovací uzol** a uzol alebo uzly z ktorých nevedú hrany ako **koncové uzly**. V našom prípade ohodnotenie hrán značí pravdepodobnosť výpadku sieťového pripojenia na hrane. Ďalej majme konštantu L ktorá predstavuje hraničné ohodnotenie pravdepodobnosti výpadku. Skratka **ZOP** nech reprezentuje zónu s obmedzeným pripojením. Vstupné uzly zóny budeme značiť ako **IN uzly** a výstupné uzly ako **OUT uzly**. Nižšie je definovanie ďalších dôležitých vlastností modelu vzhľadom na zóny obmedzeného pripojenia.

- **ZOP hrana** je hrana pre ktorú jej ohodnotenie je väčšie ako L.
- **nZOP hrana** je hrana pre ktorú jej ohodnotenie je menšie ako L.
- **ZOP** je súvislý podgraf grafu G, ktorý obsahuje len ZOP hrany.
- **IN uzol** je uzol, pre ktorý platí jedna z nasledujúcich dvoch podmienok:
  1. Uzol je štartovací uzol grafu G a zároveň z neho vystupuje hrana, ktorá je v ZOP.
  2. Z uzlu vystupuje hrana, ktorá je v ZOP a zároveň obsahuje vstupnú hranu, ktorá není v ZOP.
- **OUT uzol** je uzol, pre ktorý platí jedna z nasledujúcich dvoch podmienok:

1. Uzol je koncový uzol grafu G a zároveň z obsahuje vstupnú hranu, ktorá je v ZOP.
2. Uzol obsahuje vstupnú hranu, ktorá je v ZOP a zároveň z neho vystupuje hrana, ktorá není v ZOP.

Stavový model IoT systému

V súčasnej dobe prebieha vývoj testovacích algoritmov aj pre stavový model. Podkladom pre nasledujúci text bola rešerš z 2.5 a upresnenie požiadavok od Ing. Václava Rechtbergera, ktorý sa na vývoji týchto algoritmov podieľa.

Požiadavky na stavový model z pohľadu generovania platia totožné ako pre procesný model a sú rozšírené o nasledujúce vlastnosti:

1. Sú definované tzv. testovacie štarty, to sú uzly pre ktoré platí, že sú to koncové uzly hrán vedúcich zo štartovacieho uzlu a zároveň testovacie štarty nemôžu byť koncové uzly. Z požiadavok vyššie máme existenciu presne jedného štartovacieho uzlu a zároveň zakázané paralelné hrany, z toho vyplýva že každá hrana vedúca zo štartu definuje presne jeden testovací štart.

2. Všetky uzly s výnimkou štartovacieho uzlu a koncových uzlov môžu ale nemusia mať priradené nasledujúce príznaky *testStart*, *testEnd* a *testStartEnd*.

2.4 Oxygen

Jednou z požiadavok v zadaní je podpora platformy Oxygen, preto táto sekcia stručne zoznamuje s platformou vzhľadom na nami potrebné využitie. Oxygen je desktopová aplikácia napísaná v jazyky JAVA verzie 8, ktorá je vyvíjaná vrámci laboratória inteligentného testovania software na ČVUT FEL v Prahe.

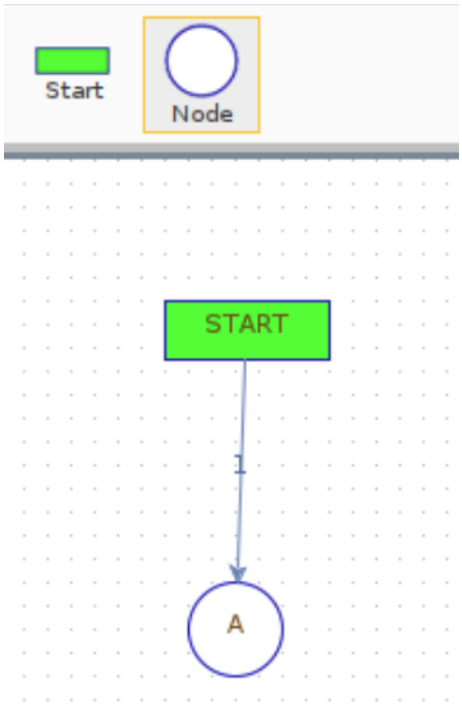
Hlavným účelom aplikácie je uľahčiť užívateľom tvorbu testovacích scenárov a to pomocou ich automatického generovania z modelu testovaného systému. Oxygen se používa taktiež pre akademické účely pri navrhovaní a overovaní nových testovacích technik. Okrem iných obsahuje aj možnosť manuálne vytvoriť model [8]. Z hľadiska tejto práce je pre nás práve táto možnosť dôležitá.

Podporované modely sú nasledujúce:

Directed Graph

Directed graf(Orientovaný graf) je graf, ktorý má dva typy uzlov a to *Start* a *Node*. Táto možnosť je vizualizovaná na obrázku 2.1.

[8] model v zmysle model testovaného systému, taktiež by išlo použiť v tomto kontexte pojem graf alebo diagram



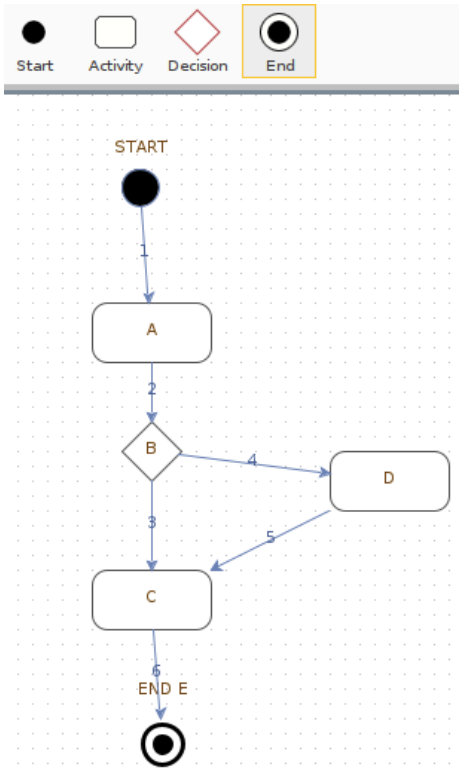
Obrázok 2.1: Ukážka *Directed Graph* v aplikácii Oxygen.

Activity Diagram

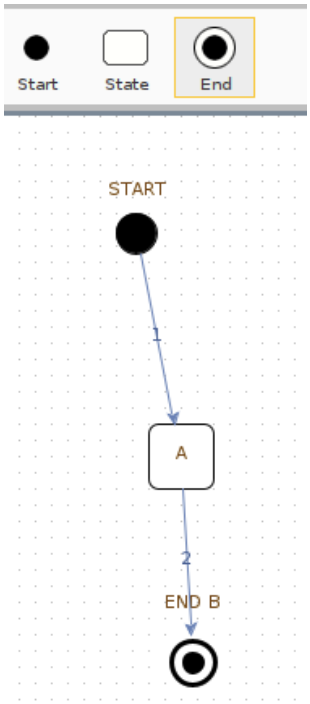
Activity diagram(diagram aktivít) je diagram, ktorý umožňuje použiť uzly typu *Start*, *Activity*, *Decision* a *End*. Ak má byť výsledný diagram validný vzhľadom na validátor v Oxygene, tak z uzlu typu *Activity* nesmie ísť viac než jedna hrana, k tomu je potrebné použiť uzol typu *Decision*. Táto možnosť je vizualizovaná na obrázku 2.2.

State Diagram

State diagram(stavový diagram) je diagram, ktorý umožňuje použiť uzly typu *Start*, *State*, a *End*. Oproti predošlým modelom sa líši taktiež vtom, že umožňuje uzlom nastaviť príznaky *testStart*, *testEnd* a *testStartEnd* z časti 2.3. Tento diagram je vizualizovaný na obrázku 2.3.



Obrázok 2.2: Ukážka *Activity Diagram* v aplikácii Oxygen.



Obrázok 2.3: Ukážka *State Diagram* v aplikácii Oxygen.

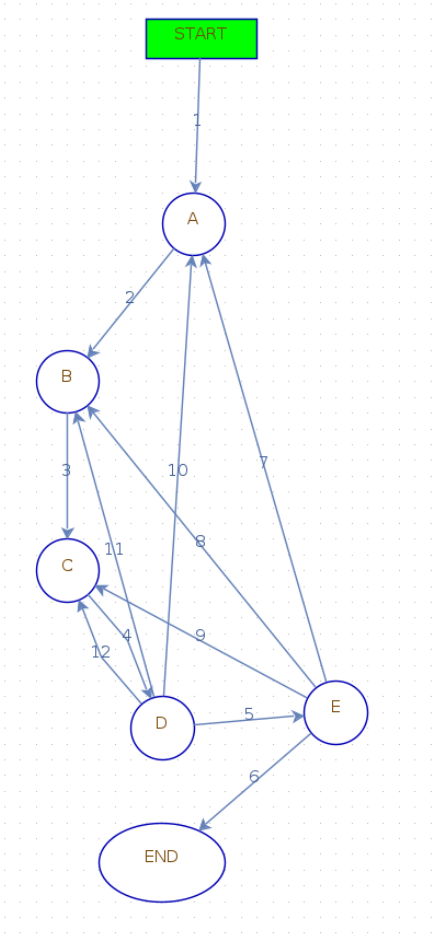
Za zmienku stojí, že aplikácia ponúka tzv. *auto arrange* možnosť, ktorá slúži na automatické rozloženie uzlov a hrán tak aby boli dobre viditeľné. Tým pádom odpadá potreba aby výsledný generátor modelov sa staral o nastavovanie súradníc jednotlivých uzlov. Všetky vyššie spomenuté modely je možné vyexportovať vo formáte *XML*.

2.5 Rešerš IoT

Hlavným cieľom tejto práce je vytvoriť generátor modelov IoT systému. Sekcia 2.3 popisuje ako by mal výsledný model vyzerať, cieľom rešerše bolo vlastnosti modelu ešte trochu upresniť, s ohľadom na návrh generátoru je dobré si položiť nasledujúce otázky:

1. Aký je približný pomer počtu hrán a počtu uzlov?
2. Aký je približný pomer počtu cyklov a počtu uzlov?
3. Aký je približný pomer počtu koncových uzlov a počtu uzlov?
4. Môže obsahovať vnorené cykly?
5. Aký je približný pomer počtu ZOP a počtu uzlov?

Motiváciou pre tieto otázky je získanie väčšej predstavy o výzore modelu. To pôjde lepšie pochopiť z nasledujúcich príkladov. Môžme si predstaviť graf, ktorý ma N uzlov, nech N je nejaké veľké číslo, indexujme uzly zaradom 1 až N. Nech uzol 1 je štartovací uzol a nemôže sa vyskytovať v cykle, uzol N nech je koncový uzol a vzhľadom nato, že z neho nemôže ísť hrana taktiež sa nemôže vyskytovat v cykle. Nech pre každý uzol s výnimkou koncového platí, že z neho vedie presne jedna hrana a to do uzlu s indexom o jedna väčším napríklad z uzlu 1 vedie hrana do uzlu 2, z uzlu 2 do 3 apod. Ak by sme chceli do takéhoto grafu dať maximum cyklov mohli by sme začať v uzle $N-1$ a viesť hranu do uzlu 2, tým by sme uzatvorili jeden cyklus, ďalej by sme mohli pokračovať tým, že by sme hranu taktiež viedli z uzlu $N-1$ ale do uzlu 3 následne do uzlu 4 apod. K získaniu ďalších cyklov nám stačí začiatočný uzol posunúť o jednotku doľava a opakovať predošlý prístup. To je ilustrované na obrázku 2.4 pre N rovné 7.



Obrázok 2.4: Príklad použitý v rešerši IoT demonštrujúci maximum možných cyklov v modeli. Model bol vytvorený v aplikácii Oxygen ako Directed Graph.

Pre obecné N by takto vyzerajúci graf mohol mať až $(n-2-1)+(n-2-2)+...+1$ cyklov.

Je ľahké vidieť, že ak by sme v predošlom príklade zvýšili počet koncových uzlov na 2 tak sa zníži maximálny počet cyklov. Vzhľadom nato, že z koncového uzlu nemôže vychádzať hrana. Ak by sa počet koncových uzlov blížil celkovému počtu uzlov, pridanie cyklu do takého grafu by sa z hľadiska generovania grafu komplikovalo(možností by bolo menej). Vzhľadom na návrh generátoru bolo potrebné mať väčšiu predstavu o tom či vyššie spomenuté príklady sú vôbec reálne. Pokiaľ takéto situácie nenastavajú je nielen zbytočné ich riešiť ale generátor, ktorý by ich zvládol by musel byť obecnejší, komplikovanejší a pravdepodobne menej efektívny na jednoduchších príkladoch. Komplikovanejší generátor by bol náchylnejší na chyby. Návrh a implementácia generátoru by sa komplikovali vzhľadom na situácie, ktoré síce sú teoretický možné z pohľadu vlastností grafu[9] no neodpovedajú realite IoT systémov.

[9] modelu

Výsledky

Počas rešerše bolo prejdených množstvo systémov. Zdroje pre jednotlivé IoT riešenia boli [16] a [17]. K vyhľadávaniu slúžili ako kľúčové slová pojmy z predošlých kapitol, napríklad kategórie využitia alebo používané technológie. Tieto riešenia boli skúmané ako z hľadiska procesov tak aj stavov. Vzhľadom nato, že v súčasnom stave sú v pokročilejšom štádiu práve algoritmy na generovanie testovacích scenárov z procesného modelu, bolo požadované niektoré systémy aj namodelovať v aplikácii Oxygen. No nie všetky systémy boli dostatočne detailne popísane aby v nich išlo určiť zóny výpadku sieťového pripojenia. Tabuľka 2.1 obsahuje namodelované systémy. V tabuľke platí nasledujúce značenie: N značí počet uzlov, E značí počet hrán, C značí počet cyklov, NE značí počet koncových uzlov, Z značí počet rôznych zón ako su definované v 2.3.

Názov	N	E	C	NE	Z
Smart Agriculture Using IoT[18]	24	28	1	2	2
Controlling Equipment[19]	20	26	2	2	2
Health Monitoring Fitness[20]	29	37	0	2	4
Parking Reservation System[21]	25	28	2	2	3
Smart Water Consumption Measurement[22]	26	32	1	2	4
Electrical Device Surveillance[23]	18	20	0	2	2
Patient Health Monitoring System[24]	22	24	3	4	2
Parking System[25]	15	17	0	2	2
Smart Homes[26]	19	21	0	2	3
Washing System of Street Lighting[27]	18	19	0	4	1
Press Shop Assembly Monitoring[28]	15	16	0	1	1

Tabuľka 2.1: Tabuľka znázorňujúca namodelované IoT systémy s ohľadom na ZOP.

Hodnoty v tabuľke vyššie taktiež zodpovedajú niektoré z otázok na začiatku tejto sekcie a to že nás generovanie krajných situácii nemusí zaujímať. Výsledný generátor teda môže mať vhodne obmedzený vstup na počet hrán, cyklov, koncových uzlov vzhľadom na celkový počet uzlov. Vnorené cykly neboli vylúčené a teda by ich generátor mal vedieť tvoriť. Ako už bolo spomenuté v časti 2.3 prejdené systémy vrámci rešerše taktiež pomohli pri stanovení požiadavkov na stavový model.

2.6 Cieľ práce

Vzhľadom na predošlý text a po diskusii so školiteľom vznikli požiadavky nato čo by mal výsledný software vedieť. Tieto požiadavky vrátane ich priorít sú popísané v nasledujúcej kapitole.

Kapitola 3
Analýza požiadavkov

Požiadavky možno podľa Neudstata a Arlowa [29] definovať ako špecifikáciu toho, čo by mal systém vedieť a za akých okolností. Funkčné požiadavky pritom určujú, aké správanie bude systém poskytovať. Kvalitatívne požiadavky popisujú ako by systém mal fungovať v kontexte jeho použitia. Funkčné požiadavky z anglického *functional requirements* budeme ďalej značiť ako FR a kvalitatívne požiadavky z anglického *non-functional requirements* ako NR.

Táto kapitola popisuje požadovaný systém z predošlej časti 2.6 pomocou vyššie spomenutých požiadavok.

3.1 Funkčné

FR1. Systém bude mať jednoduché užívateľské rozhranie pre zadanie vstupu a generovanie modelu.

FR2. Systém bude načítať parametre pre výsledný model z textového súboru.

FR3. Systém umožní načítať existujúci graf z ".xml"súboru podľa schémy definovanej platformou Oxygenom.

FR4. Systém bude schopný uložiť graf v ".xml"súbore podľa schémy definovanej platformou Oxygenom.

FR5. Systém bude kroky generovania modelu logovať do externého súboru.

FR6. V prípade nesprávnych vstupných parametrov, systém informuje užívateľa.

FR7. Z možností grafov definovaných Oxygenom, systém bude podporovať prácu s Directed graph a Activity Diagram pre procesný model.

FR8. Z možností grafov definovaných Oxygenom, systém bude podporovať prácu so State Diagram pre stavový model.

FR9. Systém umožní vytvoriť procesný model. Vstupom bude počet uzlov,

hrán, koncových uzlov a cyklov.

FR10. Systém umožní vytvoriť procesný model vrátane ZOP. Vstupom bude počet uzlov, hrán, koncových uzlov, cyklov a počet zón. Pre jednotlivé zóny to bude počet uzlov, hrán, cyklov, IN uzlov a OUT uzlov.

FR11. Systém umožní rozšíriť procesný model. Vstupom bude počet uzlov, hrán, koncových uzlov a cyklov.

FR12. Systém umožní vytvoriť stavový model. Vstupom bude počet uzlov, hrán, koncových uzlov, cyklov, počet počiatočných stavov, počet testStart, testEnd a testStartEnd.

FR13. Bez ohľadu na zvolenú možnosť generovania, v prípade že je validný vstup a algoritmus skončí úspešne bude výsledkom nový graf uložený ako samostatný súbor, nový graf musí presne spĺňať vstupné parametre.

FR14. Bez ohľadu na zvolenú možnosť, v prípade že je validný vstup a graf sa nepodarí vytvoriť bude užívateľ informovaný a nový graf nebude vytvorený.

FR15. V prípade, že to bude vstup umožňovať, systém by mal pri opakovanom volaní pre rovnaký vstup generovať rôzne modeli.

FR16. Bez ohľadu na zvolenú možnosť generovania, systém umožní výsledný model vizualizovať vo formáte *jpg*.

3.2 Kvalitatívne

NR1. Multiplatformovosť.

V našom ponímaní to znamená, že výsledná aplikácia bude fungovať na rôznych bežne používaných operačných systemoch (Microsoft Windows, Linux, Mac OS).

NR2. Možná budúca integrácia do platformy Oxygen.

NR3. Doba generovania jedného modelu maximálne v nižších jednotkách sekúnd.

3.3 Priority

Priradenie priority sa opiera o MoSCoW prioritizačnú metódu, kedy každá z požiadavok je označená jedným zo začiatočných písmenok anglických spojení Must have(M), Should have (S), Could have (C) a Won't have (W) [30].

Požiadavok	Priorita
FR1	M
FR2	M
FR3	S
FR4	M
FR5	S
FR6	M
FR7	M
FR8	M
FR9	M
FR10	S
FR11	S
FR12	M
FR13	M
FR14	M
FR15	M
FR16	C
NR1	M
NR2	C

Tabuľka 3.1: Tabuľka znázorňujúca priority jednotlivých požiadavok.

Priority sú v tabuľke 3.1.

Kapitola 4
Návrh systému

Táto kapitola popisuje voľbu použitých technológií a architektúru výsledného produktu.

4.1 Technológie

Java

Aplikácia je napísaná v programovacom jazyku Java SE 8. Java je objektovo orientovaný vyšší programovací jazyk, ktorý dedí syntax od C/C++ . Využíva sa hlavne pri webových aplikáciach, android aplikáciach ale aj pri desktopových aplikáciach. Java sa priamo nekompiluje do strojového kodu ale do tzv. "byte-code", ktorý je následne interpetovaný pomocou JVM[1] to umožňuje vykonanie bezohľadu na konkrétný operačný systém. Veľkou výhodou Javy je multiplatformovosť to spĺňa požiadavok NR1. Dôvodom pre verziu 8 bol požiadavok NR2.

Swing

Java Swing je odľahčená sada nástrojov pre grafické užívateľské rozhranie v Jave. Je súčasťou JFC[2] a obsahuje niekoľko balíkov na vývoj desktopových aplikácií v Jave. Swing obsahuje vstavané ovládacie prvky, ako sú stromy, tlačidlá, tabuľe s kartami, posúvače, panely nástrojov,... .[32] Komponenty Swingu sú napísané v Jave a sú teda nezávislé na platforme. Ďalšou možnosťou bolo použitie JavaFX, vzhľadom na jednoduchosť požadovaného užívateľského rozhrania by dobre poslúžili obidve technológie, voľba Swingu bola len osobná preferencia.

JUNG

JUNG[3] je knižnica, ktorá poskytuje bežný a rozšíriteľný jazyk pre modelovanie, analýzu a vizualizáciu údajov, ktoré môžu byť reprezentované ako

[1] Java Virtual Machine
[2] Java Foundation Classes
[3] Java Universal Network / Graph Framework

graf alebo sieť. Je napísaná v jazyku Java, ktorý umožňuje aplikáciám založeným na prostredí JUNG využívať rozsiahle vstavané funkcie rozhrania Java API, ako aj možnosti iných existujúcich knižníc Java tretích strán [31]. Vzhľadom na podporu platformy Oxygen a fakt že Oxygen umožňuje modely vizualizovať a taktiež umožnuje tzv. auto-arrange, nebola nutnosť implicitne nastavovať súradnice jednotlivým uzlom vo výslednom modeli. JUNG bol vhodný pretože túto možnosť umožňuje taktiež. Patrí síce k starším no dobre zdokumentovaným frameworkom.

XML

XML[4] je jednoduchý textový formát na reprezentáciu štruktúrovaných informácií ako dokumenty, údaje, konfigurácie, knihy, transakcie, faktúry a oveľa viac. Bol odvodený zo staršieho štandardného formátu s názvom SGML (ISO 8879), aby bol vhodnejší na použitie na webe. Dnes je jedným z najbežnejšie používaných formátov na zdieľanie štruktúrovaných informácií [33]. Požíva ho taktiež Oxygen.

Apache Maven

Maven je nástroj na riadenie softvérových projektov. Na základe konceptu projektového objektového modelu(POM) môže Maven spravovať zostavenie, vykazovanie a dokumentáciu projektu [34].

Apache Commons IO

Je knižnica funkcionalít ktoré pomáhajú pri práci s čítaním/zapisovaním z/do súborov [35].

[4]Extensible Markup Language

4.2 Popis architektúry

K popisu architektúry boli zvolené *4+1* pohľady podľa P. Kruchtena [36]. Jednotlivé pohľady sú logický, procesný, implementačný a fyzický pohľad. Plus jedna predstavuje prípady použitia alebo scenáre. Logický pohľad sa zameriava hlavne na funkčné požiadavky, ukazuje ako je systém rozložený do jednotlivých tried alebo skupín tried aby ich splnil a asociácie týchto tried. Procesný pohľad sa zaoberá dynamickými aspektmi systému, vysvetľuje systémové procesy a ich komunikáciu, taktiež sa zameriava na správanie systému za behu. Implementačný pohľad bližšie popisuje jednotlivé komponenty systému. Fyzický pohľad sa zaoberá topológiou softvérových komponent na fyzickej vrstve a prepojeniami medzi nimi.

Prípady použitia

Diagram na obrázku 4.1 zobrazuje základné prípady použitia výslednej aplikácie.

Logický pohľad

Diagram na obrázku 4.2 predstavuje logický pohľad na architektúru. Sú na ňom skupiny tried a prepojenia medzi nimi. Jednotlivé skupiny sú:

1. užívateľské rozhranie - triedy, ktoré sa starajú o UI aplikácie
2. služby - sú využívané užívateľským rozhraním a volajú jednotlivé hlavné funkcionality k dosiahnutiu funkčných požiadavok
3. výstup - starajú sa o výstup generovania ako vypísanie modelu alebo vykreslenie obrázku modelu
4. vstup - starajú sa o validáciu vstupného textového súboru a jeho hodnôt, vstup sa načítava z FS teda využívajú prácu so súbormi
5. generátor - triedy ktoré obsahujú samotné algoritmy na generovanie požadovaných modelov
6. práca so súbormy
7. vstup doména - triedy reprezentujúce jednotlivé užívateľské vstupy pre generovanie modelov
8. model doména - navrhnuté dátové štruktúry používane pri generovaní modelu
9. logovanie

Procesný pohľad

Pre popísanie procesného pohľadu bol zvolený *UML Diagram Aktivít*, ten je znázornený na obrázku 4.3. Ukazuje sled aktivít ktoré je potrebné vykonať k dosiahnutiu funkčných požiadavok týkajúcich sa samotného generovania modelu.

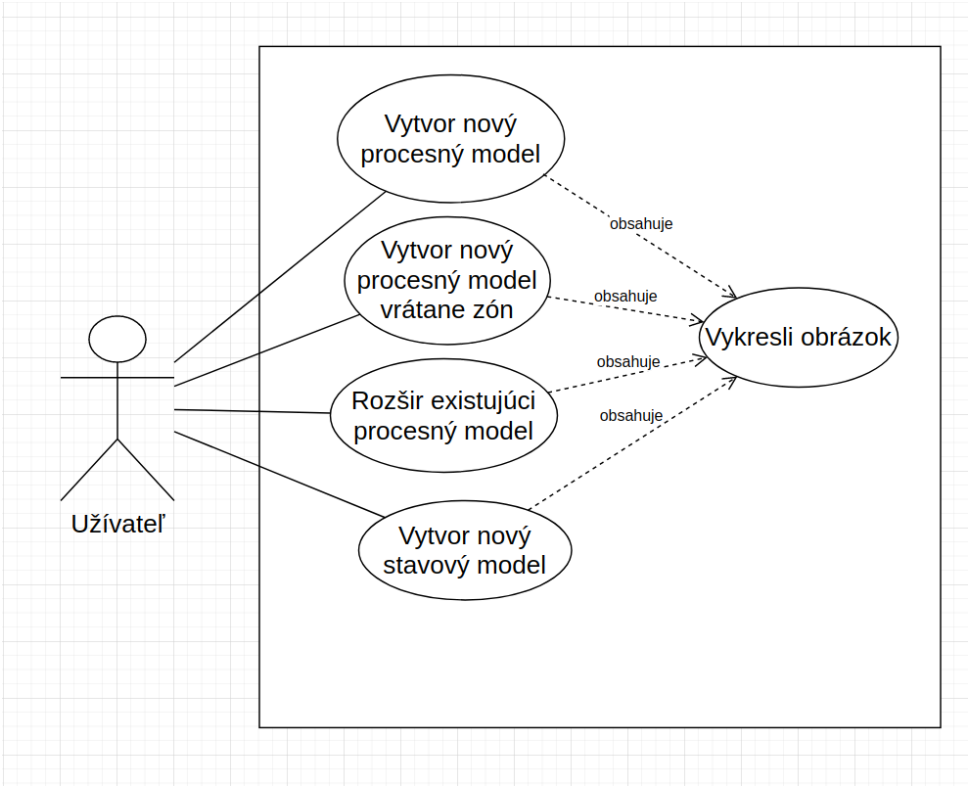
Implementačný pohľad

Samotnú implementáciu bližšie ilustruje *UML Diagram Balíkov*, ktorý je znázornený na obrázku 4.4. Aplikácia je členená do dvoch hlavných balíkov a to *ui* a *business*. Balík *ui* obsahuje logiku užívateľského rozhrania, balík *business* by išlo stručne popísať ako back-end aplikácie. Balík *business* je ďalej členený do nasledujúcich balíkov:

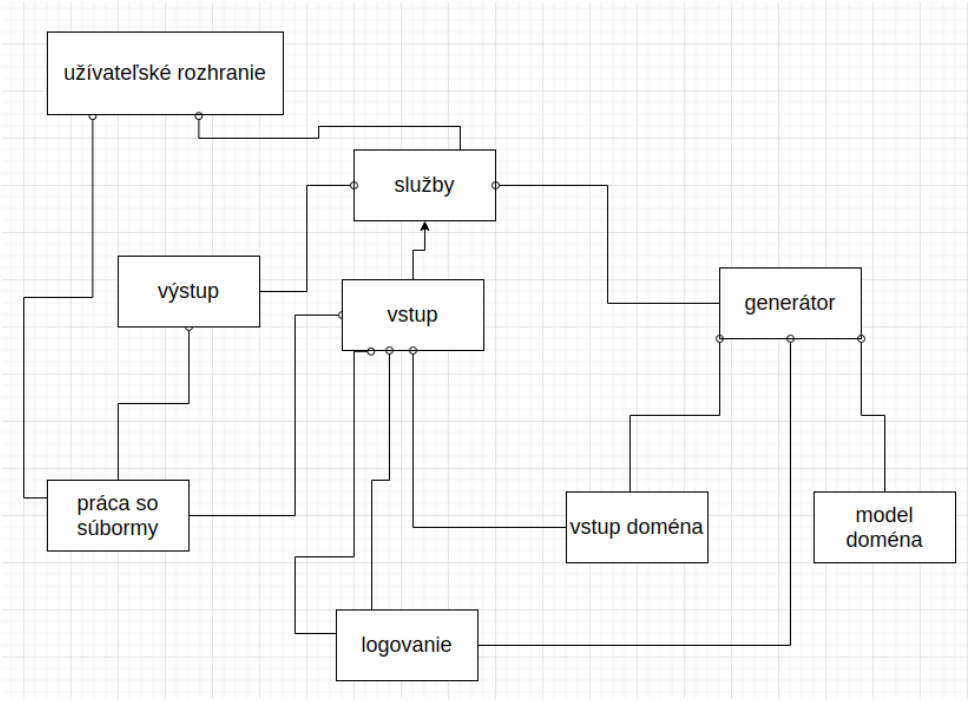
- service - obsahuje jednotlivé služby, ktoré volajú ostatnú logiku aplikácie k dosiahnutiu funkčných požiadavok
- utils - obsahuje pomocné triedy napríklad na prácu načítanie/vypísanie modelu
- generator - obsahuje samotnú logiku generovania jednotlivých modelov
- input - triedy, ktoré sa starajú o načítanie a validáciu vstupu
- oxygen_model - konvertuje vygenerovaný model do formátu *XML* podľa požiadavok *Oxygenu*
- model_domain - obsahuje dátové štruktúry používané pri generovaní modelu, tie sú taktiež potrebné pri konverzii do *XML*
- input_domain - obsahuje triedy reprezentujúce užívateľský vstup

Fyzický pohľad

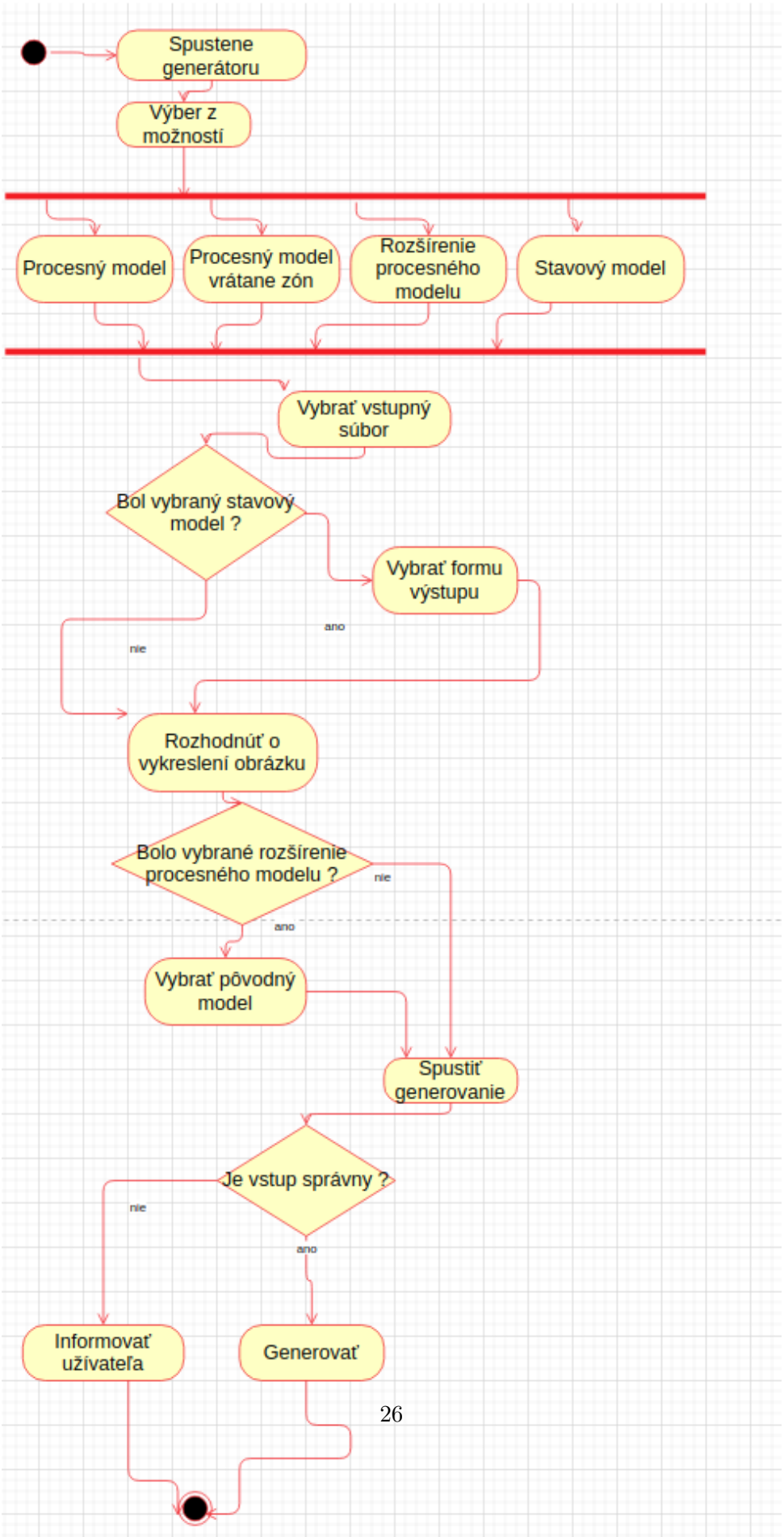
K fyzického pohľadu na architektúru bol použitý *UML Diagram Nasadenia*, ktorý je na obrázku 4.5. K spusteniu aplikácie(Generator.jar) je potrebné mať počítač s podporou Java SE 8.



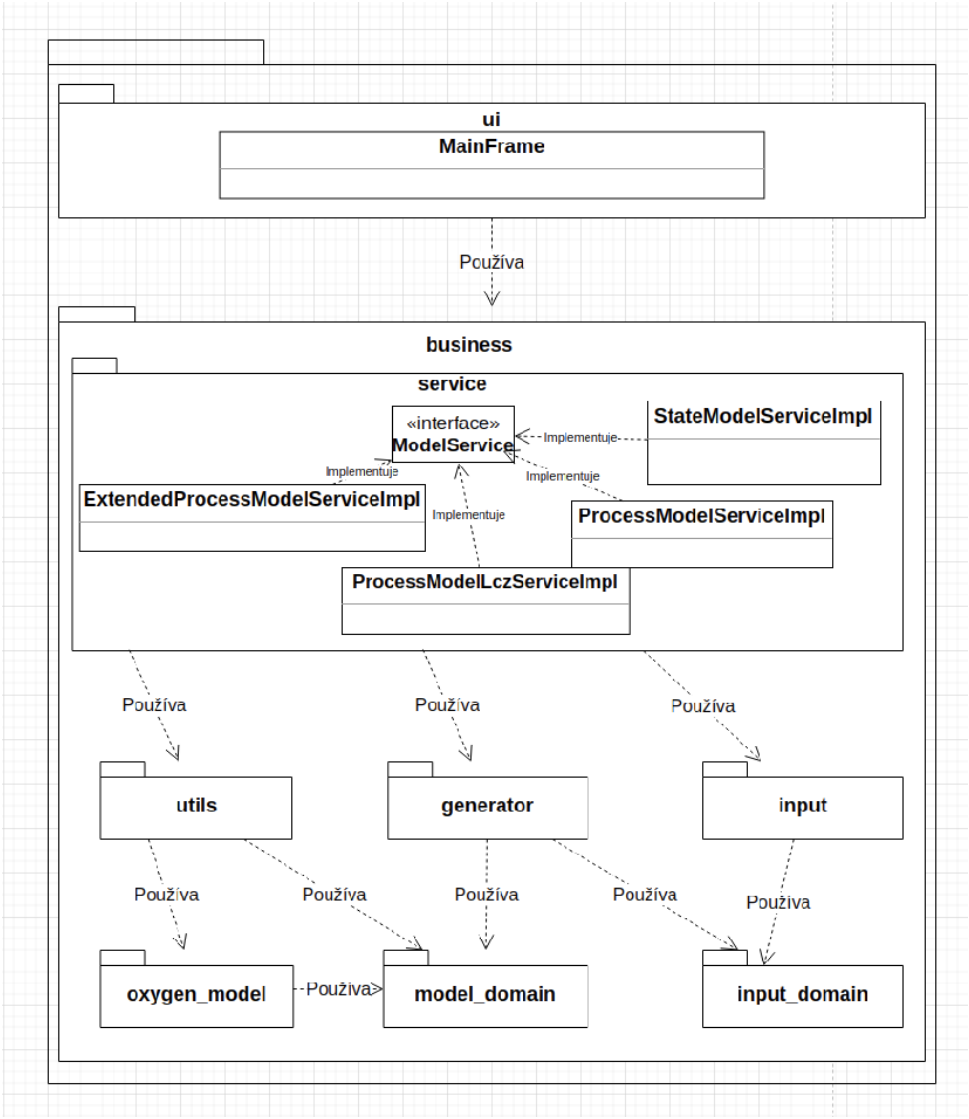
Obrázok 4.1: *Diagram prípadov použitia.*



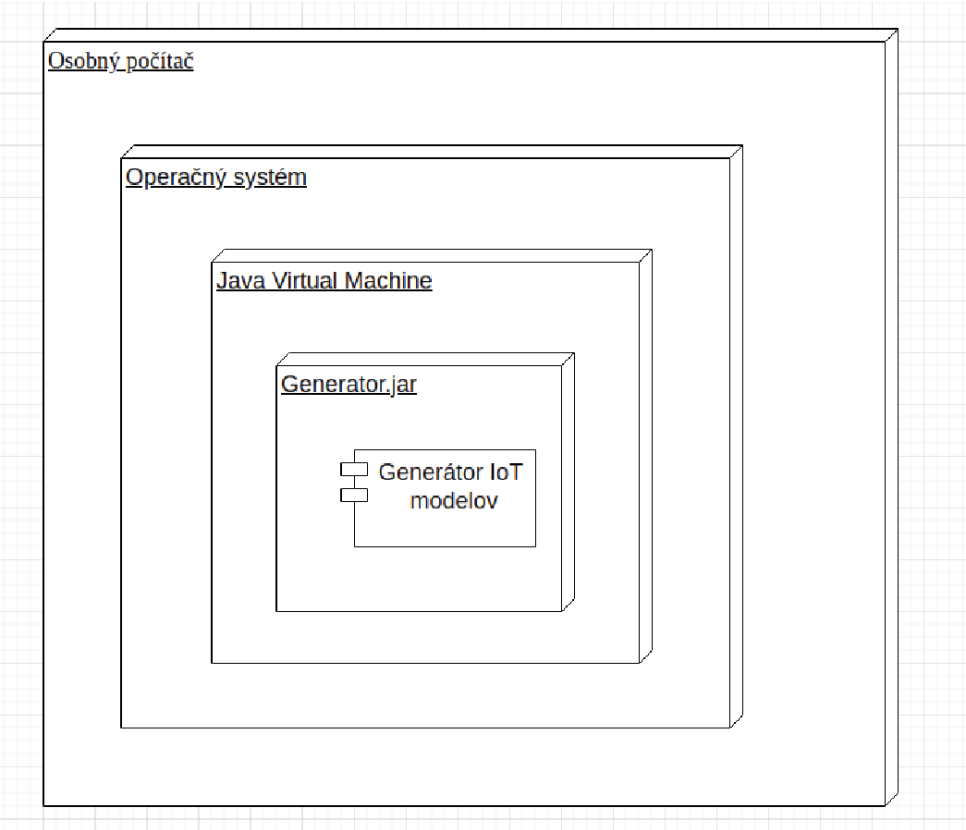
Obrázok 4.2: *Diagram znázorňujúci skupiny tried.*



Obrázok 4.3: *Diagram aktivít.*



Obrázok 4.4: *Diagram balíkov.*



Obrázok 4.5: *Diagram nasadenia.*

Kapitola 5
Popis riešenia

Táto kapitola popisuje riešenie problematiky, zameriava sa prioritne na popis vstupu a generovania jednotlivých modelov, končí ukážkou užívateľského rozhrania. V ďalšom texte bude vo väčšom detaile popísané generovanie procesného modelu. Dôvodom nato je, že ostatné možnosti generátoru: stavový model, procesný model so ZOP a rozšírenie procesného modelu v silnej miere využívajú a vychádzajú práve z generovania procesného modelu. Pri navrhovaní postupu ako aj obmedzení na vstupné údaje boli využité výsledky z rešerše IoT 2.5 a konzultácie so školiteľom. Dôležitým pozorovaním z tabuľky 3.1 je že výsledný model je len minimálne väčší než minimálny[1] možný model z 2.3. Preto obecný prístup ku generovaniu je tento minimálny model na začiatku inicializovať a potom pridať k modelu potrebný zvyšok, inicializáciu minimálneho modelu podrobnejšie popisuje *Inicializácia základného modelu* v 5.1.2.

5.1 Procesný model

Táto sekcia popisuje vytvorenie procesného modelu podľa zadaného vstupu.

5.1.1 Obmedzenia na vstup

Ako popisuje požiadavok FR9 vstupom je počet uzlov, počet hrán, počet koncových uzlov a počet cyklov. Na vstup boli stanovené nasledujúce obmedzenia:

1. Počet uzlov(*nodesCount*)
   - Minimálny počet uzlov musí byť aspoň 5.
   - Maximálny počet uzlov je 500.
2. Koncové uzly(*endNodesCount*)
   - Minimálny počet koncových uzlov musí byť aspoň 1.
   - Maximálny počet koncových uzlov je $nodesCount - 2$.

[1] minimálny v zmysle minimum hrán aby bol graf súvislý, existencia štartovacieho uzlu, koncového uzlu

3. Počet cyklov(*cyclesCount*)
   - Minimálny počet cyklov je 0.
   - Maximum pre počet cyklov označme ako $maxCyclesCount$, platí: $maxCyclesCount = nodesCount - endNodesCount - 1$.
4. Počet hrán(*edgesCount*)
   - Minimálny počet hrán označme ako $minEdgesCount$, platí: $minEdgesCount = nodesCount - 1 + cyclesCount$.

     Podmienka $nodesCount - 1$ vyplýva z faktu, že výsledný model má byť súvisly graf, $cyclesCount$ hrán potrebujeme na uzatvorenie cyklov.
   - Maximum pre počet hrán označme ako $maxEdgesCount$, platí:

     $normalNodes = nodesCount - endNodesCount - 1$

     $otherEdges = \sum_{i=1}^{normalNodesCount-2} x_i - \sum_{i=1}^{cyclesCount-1} y_i$

     $maxEdgesCount = minEdgesCount + normalNodes - 1 + otherEdges$.

     K odôvodneniu týchto vzťahov je potrebné mať znalosti o fungovaní generátoru, preto je až na konci ďalšej sekcie.

5.1.2 Popis generátoru

Táto sekcia popisuje ako funguje generovanie procesného modelu. Nižšie je znázornený pseudokód 1 algoritmu, ktorý ilustruje hlavnú logiku samotného generovania, využívajú sa v ňom označenia zo sekcie 5.1.1.

Inicializácia základného modelu

Generátor začína tým, že vytvorí pole uzlov o veľkosti nodesCount. V jave sú prvky poľa indexované od nuly, ďalej sa budeme držať tohto prístupu. Prvok s indexom nula sa označí ako štartovací uzol a prvok s indexom *nodesCount - 1*(teda posledný prvok poľa) ako koncový uzol. Ďalej platí že pre všetky uzly poľa s výnimkou posledného vedie hrana do prvku s indexom o jedna väčším. Teda sa pridá *nodesCount - 1* hrán. Po tomto už máme presne chcený počet uzlov a taktiež je splnená súvislosť grafu, existencia štartovacieho a minimálne jedného koncového uzlu.

Pridanie cyklov

Ďalšou výhodou vyššie spomenutého prístupu je jednoduché pridávanie cyklov, po takejto inicializácii vytvorenie nového cyklu v modeli znamená pridať

Algorithm 1 Generátor procesného modelu

Input Zvalidovaný vstup od užívateľa.
Output Procesný model odpovedajúci zadanému vstupu.

1: $maxCycleSize \leftarrow nodesCount - 2$
2: $iteration \leftarrow 0$
3: $iterationForCycleSize \leftarrow 0$
4: $limitForIteration \leftarrow nodesCount$
5: **while** true **do**
6: $iteration \leftarrow iteration + 1$
7: $iterationForCycleSize \leftarrow iterationForCycleSize + 1$
8: **if** $maxCycleSize > 1$ **and** $iterationForCycleSize = limitForIteration$ **then**
9: $maxCycleSize \leftarrow \left\lfloor \frac{maxCycleSize}{2} \right\rfloor$
10: $iterationForCycleSize \leftarrow 0$
11: $nodesArray \leftarrow$ Inicializácia základného modelu.
12: Pridanie cyklov. ▷ modifikuje *nodesArray*
13: Oprava identifikátorov cyklov. ▷ modifikuje *nodesArray*
14: Vytvorenie koncových uzlov. ▷ modifikuje *nodesArray*
15: **if** Vytvorenie koncových uzlov nebolo úspešné. **then**
16: Pokračuj ďalšou iteráciou while.
17: Pridanie zostávajúcich hrán. ▷ modifikuje *nodesArray*
18: **if** pridanie zostavajúcic hrán nebolo úspešné **then**
19: Pokračuj ďalšou iteráciou while.
return *nodesArray*
20:

hranu ktorá začína v uzly s indexom i a končí v uzly s indexom j kde platí, že j je menšie ako i.

Pridanie hrany, ktorá uzatvára cyklus funguje následovne:

1. Vytvorí sa zoznam potenciálnych indexov v ktorých hrana môže začínať, to sú prirodzené čísla od 2 do nodesCount - 2, označme tento zoznam ako startIndexList.

2. Vyberie sa náhodne jeden prvok zo startIndexList, označme ako startIndex.

3. Pre daný startIndex sa vytvorí zoznam potenciálnych indexov v ktorých hrana môže končiť, to sú prirodzené čísla od max(startIndex - maxCycleSize, 1) po startIndex - 1, označme ako endIndexList.

4. Zoznam endIndexList sa náhodne roztriedi.

5. Z endIndexList sa vyberie prvý prvok, označme ako endIndex.

6. V prípade, že neexistuje hrana z uzlu s indexom startIndex do uzlu s indexom endIndex, preskakuje sa na bod 7, v opačnom prípade sa endIndex odmaže z endIndexList a môže nastať jedna z nasledujúcich situácií:
   - endIndexList nie je prázdny, potom sa pokračuje bodom 5
   - endIndexList je prázdny, potom sa pokračuje bodom 2 a startIndex sa odmaže zo startIndexList
7. Pridanie hrany z uzlu s indexom startIndex do uzlu s indexom endIndex, všetky uzly v novo pridanom cykle dostanú identifikator cyklu, ktorý sa pre ďalšie iterácie inkrementuje.

Postup vysvetlený vyššie sa opakuje kým nie je pridaných cyclesCount cyklov, ako je ukázané vyššie tento postup je konečný a vzhľadom na obmedzenie na počet cyklov nemôže skončiť neúspechom.

Vytvorenie koncových uzlov

Vzhľadom nato, že z koncového uzlu nemôžu ís hrany vyplýva, že koncový uzol nemôže byť v cykle. Vytvorenie koncového uzlu funguje tak, že sa vyberie uzol ktorý nie je koncový a nie je v cykle. Tento uzol sa označí ako koncový a hrany, ktoré z neho vychádzajú budu vychádzať z iného uzlu s nižším indexom, ten sa náhodne vyberie, fakt že ma nový štart pre hrany nižší index je dôležitý aby vlastnosti grafu zostali zachované a nevytvorili sme nechtiac nové cykly.

Tento postup narozdiel od pridania cyklov môže skončiť neúspechom, dôvodom nato je fakt, že sa nenájde dostatok uzlov, ktoré nie sú v cykle inak povedané cykly, ktoré sme pridali v predošlom bode budú príliš veľké. V pseudokóde je ukázané riešenie tohto problému, vyvolá sa ďalšia iterácia generovania a po fixnom počte iterácii sa maximálna veľkosť cyklu zníži až sa postupne vie znížiť len na spätné hrany(odpovedá veľkosti jedna). Obmedzenia na vstup sú definované tak, aby na sebe maximálny počet cyklov a maximálny počet koncových uzlov záviseli, z toho mame istotu ze zníženie maximálnej veľkosti cyklu bude stačiť nato aby bolo možné v konečnom čase pridať chcené koncové uzly. Inak povedané vďaka jednotkovým cyklom určite zostane dosť voľných uzlov pre koncové uzly.

Pridanie zostávajúcich hrán

V prípade pridávania cyklov sme pridávali hrany z prava do ľava to nám zaručovalo vytvorenie cyklu, v prípade normálnych hrán, ktoré nemajú cyklus uzavrieť a ani nesmú pretože chcený počet cyklov už máme sa hrany pridávajú z ľava do prava. Teda počiatočný uzol hrany má nižší index ako koncový uzol

hrany.

Vzhľadom na už existujúce cykly môžu avšak nastať dva problémy, prvým je pridanie hrany vo vnútri cyklu to by vytvorilo nový cyklus , druhým je pridanie hrany medzi dvomi susednými cyklami, to by opäť mohlo končiť vytvorením nechceného cyklu. K ilustrovaniu týchto problémov využijeme obrázok5.1, na ktorom môžme vidieť nasledujúce dva cykly:

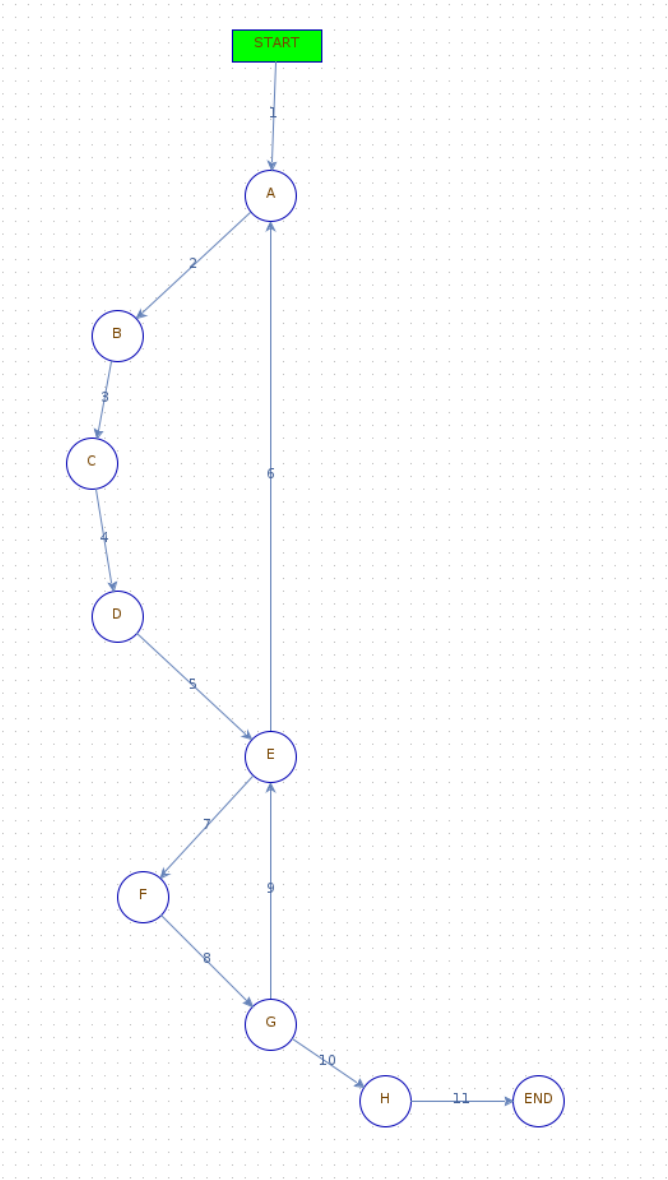
1. $A, 2, B, 3, C, 4, D, 5, E, 6, A$

2. $E, 7, F, 8, G, 9, E$

niekto by mohol namietať, že je tam aj tretí cyklus a to:

$A, 2, B, 3, C, 4, D, 5, E, E, 7, F, 8, G, 9, E, 6, A$
ale to by bolo v rozpore s definíciou z časti 2.1 pretože sa tu opakuje uzol E dvakrát.

Vidíme z obrázku že pridanie hrany napríklad z uzlu C do uzlu H cyklus nevytvorí, zatiaľ čo pridanie z uzlu C do uzlu F áno, nový cyklus by bol: $C, nováHrana, F, 8, G, 9, E, 6, A, 2, D, 3, C$.



Obrázok 5.1: Príklad grafu.

V pseudokóde je tento problém riešení pomocou metódy *Oprava identifikátorov cyklov*, pri pridávaní cyklov sme uzli v cykle označili identifikátorom. Pre náš príklad si môžme predstaviť, že uzly z prvého cyklu nesú informáciu o tom že sú v cykle s indexom 1, uzly druhého cyklu zas nesú informáciu o tom, že sú v cykle s indexom 2, uzol E avšak nesie informáciu o oboch cykloch, *Oprava identifikátorov cyklov* rieši práve tento problém a to tak, že pre každý cyklus prejde všetky jeho uzly a informácie o indexoch cyklov vloží do množiny potom prejde uzly ešte raz a nastaví výslednú množinu. S touto informáciou počíta pridávanie nových hrán.

Pridanie novej hrany funguje nasledujúco:

1. Vytvorí sa zoznam potenciálnych indexov v ktorých hrana môže začínať, to sú prirodzené čísla od 0 do nodesCount - 3, označme tento zoznam ako startIndexList.

2. Vyberie sa náhodne jeden prvok zo startIndexList, označme ako startIndex. V prípade že uzol so startIndex je koncový tak sa startIndex odmaže zo zoznamu kandidátov a vyberá sa znovu. V prípade, že je startIndexList prázdny algoritmus končí neúspechom.

3. Vytvorí sa zoznam potenciálnych koncových uzlov označme ako endIndexList, to sú prirodzené čísla od startIndex + 2 po nodesCount - 1. Zdôraznime, že hrana z uzlu s indexom startIndex do uzlu s indexom startIndex + 1 už vedie kvôli inicializácii modelu.

4. Vyberie sa jeden prvok z endIndexList, označme ako endIndex. V prípade, že je endIndexList prázdny startIndex sa odmaže zo startIndexList a pokračuje sa bodom 2.

5. Tu nastáva overenie startIndex a endIndex, môžu nastať nasledujúce situácie:

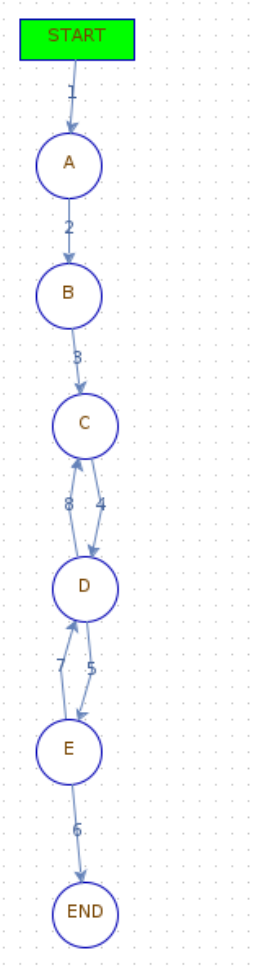
   - uzol s indexom startIndex je štartovací uzol modelu a uzol s indexom endIndex je koncový uzol
   - hrana medzi uzlamy s indexami startIndex a endIndex už existuje
   - jedná sa o susedné cykly alebo by novo pridaná hrana bola vo vnútri cyklu

   ak nastane jedna z vyššie spomenutých situácií tak sa endIndex odmaže a pokračuje sa bodom 4 v opačnom prípade sa pridá nová hrana a algoritmus končí.

   Rovnako ako vytváranie koncových uzlov aj pridávanie hrán vie skončiť neúspechom dôvodom môžu byť buď príliš veľké cykly alebo susedné cykly, ktoré vyústia v nedostatok možností na pridanie požadovaných hrán. Rovnako ako v predošlom prípade je riešením zníženie maximálnej veľkosti cyklu v kombinácii s vhodne obmedzeným maximom na počet hrán.

5.1.3 Konečnosť postupu

Vzhľadom na obmedzenie vstupu a vymazávanie nevalidných možnosti pri pridávaní cyklov, tvorbe koncových uzlov alebo pridávaní nových hrán je zabezpečená konečnosť generovania. Zostáva vysvetliť obmedzenie na maximálny počet hrán. To ukážeme na nasledujúcom príklade. Model je ilustrovaný na obrázku5.2. Ďalej sa budeme držať pojmov definovaných v 5.1.1.



Obrázok 5.2: Príklad grafu.

Máme model o veľkosti 7 uzlov, kde jeden uzol je štartovací a jeden je koncový. Z inicializácie máme 6 hrán, kvôli pridaniu dvoch cyklov ďalšie 2 hrany, to odpovedá *minEdgesCount*. Zo štartu môžu viesť ešte hrany do uzlov *B, C, D, E* to je obecne *normalNodes - 1*. Kde *normalNodes* značí uzly ktoré nie sú koncové a ani štartovací uzol, mínus 1 pretože odpočítavame uzol do ktorého už vedie jedna hrana z inicializácie modelu.

V našom prípade *normalNodes* je rovné 5. Z uzlu A môžu viesť ďalšie hrany do uzlov *C, D, E*. Z uzlu B je to zas do uzlov *D, E*. Vidíme, že sa počet zmenšil o jednu hranu, toto platí obecne. Napríklad ak by medzi štartom a uzlom A bol nejaký uzol mohli by sme z neho pridať hrany do uzlov *B, C, D, E*. Z uzlov *C a D* už ďalšie hrany nepridáme. Maximum je teda 5, čo obecne odpovedá formule pre *otherEdges*. Ak by *cyclesCount* bolo 0, tak by sme mohli pridať ešte jednu hranu a to z uzlu C do E. Odpočítanie je zahrnuté vo formuli pre *otherEdges*.

Z vyššie spomenutého vychádza *maxEdgesCount* rovné 17. O to aby to šlo

dosiahnuť sa v algoritme stará zmenšovanie maximálnej veľkosti cyklu. To že k tejto situácii raz dôjde zabezpečuje náhodné vyberanie pri tvorbe modelu v kombinácii s nekonečným *while* cyklom.

5.2 Stavový model

Vzhľadom nato, že z pohľadu generovania je stavový model veľmi podobný procesnému ako ukazuje sekcia 2.3 bol pre generovanie použitý skoro totožný postup ako v predošlom prípade. Z toho dôvodu sa táto sekcia silne opiera o predošlú, ako čo sa týka požiadavok na vstup tak aj samotného generovania. Namiesto kopírovania predošlej sekcie, tu budú len rozdiely oproti procesnému modelu.

5.2.1 Vstup

Ako popisuje požiadavok FR12 vstupom je počet uzlov, hrán, koncových uzlov, cyklov, počet počiatočných stavov, počet testStart, testEnd a testStartEnd. Obmedzenia na vstup platia tak ako sú definované v 5.1.1 a ďalej platí:

1. Počet počiatočných stavov
   - Minimálny počet počiatočných stavov je 1.
   - Maximum pre počet počiatočných stavov označme ako $maxStartStatesCount$, platí:
     $maxStartStatesCount = nodesCount - endNodesCount - 1$.
2. Počet testStart, testEnd a testStartEnd
   - Minimálny počet testStart, testEnd a testStartEnd je nula.
   - Označme maximum pre počet všetkých príznakov testStart, testEnd a testStartEnd ako $maxTestFlags$, platí:
     $maxTestFlags = nodesCount - endNodesCount - 1$.
3. Počet hrán(edgesCount)
   - Minimálny počet hrán označme ako $minEdgesCount$, platí:
     $minEdgesCount = nodesCount$ - 1 + $cyclesCount$ + $startStates-Count$ - 1 .

     Oproti procesnému modelu je tu jediný rozdiel v pridaní hrán do počiatočných stavov, to je síce startStatesCount no jedna hrana z nich je už obsiahnutá v nodesCount - 1, to je z dôvodu, že jeden testState je vytvorení už v inicializácii základného modelu.
   - Maximum pre počet hrán označme ako $maxEdgesCount$, platí:

     $otherEdges = \sum_{i=1}^{normalNodesCount-2} x_i - \sum_{i=1}^{cyclesCount-1} y_i$

$$maxEdgesCount = minEdgesCount + otherEdges\ .$$

Oproti procesnému modelu tu nie je normalNodesCount to je z dôvodu, že zo štartu ide presne startStatesCount hrán, ktoré už sú obsiahnuté v minEdgesCount.

5.2.2 Popis generátoru

Pri generovaní stavového modelu platí algoritmus z predošlej sekcie 1, s nasledujúcimi tromi rozdielmi:

1. V prípade *Pridanie zostávajúcich hrán* 5.1.2 sa pre zoznam potenciálnych indexov(startIndexList), kde hrana môže začínať posúva ľavá hranica z 0 na 1. To je dané kvôli presne danému počtu testovacích stavov.
2. Potom čo je *Pridanie zostávajúcich hrán* dokončené sa pridajú testovacie stavy. To znamená len pridať hrany vedúce zo štartovacieho uzlu do uzlov, ktoré nie sú koncové.
3. Následne sa niektoré z uzlov, ktoré nie sú koncové a ani štartovací uzol nastavia na testStart, testEnd a testStartEnd. Riešené je to tak, že sa indexy poľa(z inicializácie modelu) s výnimkou nultého(to je štart) dajú do zoznamu a náhodne roztriedia, nasleduje cyklus, ktorý pridá testStart uzlov. To funguje tak, že sa uzol pre vybraný index skontroluje či nieje koncový a v prípade, že je, tak sa len vyvolá ďalšia iterácia a neinkrementuje sa čítač v opačnom prípade sa uzol nastaví na testStart. V obidvoch situáciach sa uzol zo zoznamu odstráni. Obdobne to funguje pre nastavenie testEnd a testStartEnd. Stále sa avšak pracuje s rovnakým zoznamom aby sme mali istotu že sa nepoužijú rovnaké uzly dvakrát.

Všetky vyššie spomenuté zmeny oproti generovaniu procesného modelu musia skončiť úspešne. To je zabezpečené pomocou vhodného obmedzenia na vstup. Z hľadiska konečnosti postupu sa oproti generovaniu procesného modelu nič nemení.

5.3 Procesný model so ZOP

Táto sekcia popisuje ako funguje generovanie procesného modelu s ZOP. Ako už bolo spomenuté v 2.3 pod pojmom ZOP myslíme zóny obmedzeného pripojenia, v ďalšej časti budeme nato niekedy skrátene odkazovať pod pojmom *zóna*.

5.3.1 Vstup

Ako je uvedené v 2.3 ZOP nie je nič iné ako podgraf celého grafu[2]. V prípade, že máme generovať model so zónami ide vstup rozdeliť na popis jednotli-

[2] v našom prípade graf sa rovná model

vých zón a zvyšku modelu. Pre jednotlivé zóny chceme zadať presný počet uzlov, počet hrán, počet cyklov, počet vstupných uzlov a počet výstupných uzlov. Označme tieto vstupy zaradom ako zNodesCount, zEdgesCount, zCyclesCount, InNodesCount a OutNodesCount. Vzhľadom na generovanie jednotlivých zón boli stanovené nasledujúce obmedzenia:

1. zNodesCount
   - Minimálna hodnota pre zNodesCount je rovná 3.
   - Maximálna hodnota pre zNodesCount je rovná 50.
2. zCyclesCount
   - Minimálna hodnota pre zCyclesCount je rovná 0.
   - Maximálna hodnota pre zCyclesCount je rovná zNodesCount - 1.
3. zEdgesCount
   - Minimálny počet hrán označme ako $minEdgesCount$, platí: $minEdgesCount = zNodesCount - 1 + zCyclesCount$.
   - Maximum pre počet hrán označme ako $maxEdgesCount$, platí:

     $otherEdges = \sum_{i=1}^{zNodesCount-2} x_i - \sum_{i=1}^{cyclesCount-1} y_i$

     $maxEdgesCount = minEdgesCount + otherEdges$.
4. InNodesCount
   - Minimálna hodnota pre InNodesCount je rovná 1.
   - Maximálna hodnota pre InNodesCount je rovná zNodesCount.
5. OutNodesCount
   - Minimálna hodnota pre OutNodesCount je rovná 1.
   - Maximálna hodnota pre OutNodesCount je rovná zNodesCount.

Okrem samotných zón zostáva zadať vstup pre zvyšok modelu. Počet zvyšných uzlov, hrán, koncových uzlov, cyklov a nakoniec celkový počet zón. Tieto vstupy označme zaradom ako nodesCount, edgesCount, endNodesCount, cyclesCount a zonesCount. Z pohľadu ďalších obmedzení potrebujeme zadefinovať dva pojmy a to totalInNodesCount a totalOutNodesCount, kde totalInNodesCount je súčet všetkých InNodes cez všetky zóny, obdobne totalOutNodesCount je súčet všetkých OutNodes cez všetky zóny. Platia nasledujúce obmedzenia:

1. nodesCount

- Minimálna hodnota pre nodesCount je rovná: totalInNodesCount + totalOutNodesCount + 1 .
- Maximálna hodnota pre nodesCount je rovná 500.

2. edgesCount
   - Pre minimálnu hodnotu edgesCount platí, že je rovná: nodesCount - 1 - zonesCount + totalInNodesCount + totalOutNodesCount
   - Maximum pre počet hrán označme ako $maxEdgesCount$, platí:

     $$normalNodes = nodesCount - endNodesCount - 1$$

     $$otherEdges = \sum_{i=1}^{normalNodesCount-2} x_i - \sum_{i=1}^{cyclesCount-1} y_i$$

     $$maxEdgesCount = minEdgesCount + normalNodes - 1 + otherEdges\ .$$

3. endNodesCount
   - Minimálna hodnota pre endNodesCount je rovná 1.
   - Maximálna hodnota pre endNodesCount je rovná: nodesCount - 1 - totalInNodesCount.
4. cyclesCount
   - Minimálna hodnota pre cyclesCount je rovná 0.
   - Maximálna hodnota pre cyclesCount je rovná: nodesCount - 3.
5. zonesCount
   - Minimálna hodnota pre zonesCount je rovná 1.

5.3.2 Popis generátoru

Táto sekcia popisuje ako funguje generovanie procesného modelu so ZOP. Hlavná postupnosť jednotlivých krokov je znázornená na pseudokóde 2.

Vytvorenie úsekov

Pod pojmom úsek, myslíme súvislý podgraf, s podobnými vlastnosťami ako v prípade *Inicializácia základného modelu* pri generovaní procesného modelu. Nech úsek sa skladá z M uzlov, potom má presne M - 1 hrán. Uzly sú postupne indexované a platí, že hrana vedie z uzlu s nižším indexom do uzlu s vyšším indexom, s výnimkou posledného uzlu z ktorého hrana nevedie. Celkový počet úsekov je zonesCount + 1 a dokopy majú nodesCount uzlov. Teda zóny plus úseky dokopy tvoria celý model.

Majme prirodzené číslo i pre ktoré platí $i \in [1, zonesCount]$. ZOP_i nech značí i-tú zónu a úsek_i nech značí i-tý úsek, potom platí nasledujúce:

- pre úsek$_1$ je minimálny počet uzlov InNodesCount zóny ZOP$_1$
- pre úsek$_{zonesCount+1}$ je minimálny počet uzlov OutNodesCount zóny ZOP$_{zonesCount}$
- pre každý úsek$_i$ kde $i \in [2, zonesCount - 1]$ je minimálny počet uzlov rovný súčtu InNodesCount zóny ZOP$_{i-1}$ a OutNodesCount zóny ZOP$_i$

Zvyšok uzlov sa náhodne rozdelí medzi úsekmi.

Vytvorenie zón

Ako už bolo vyššie spomenuté finálny počet zón je zonesCount, každá jedna z týchto zón je súvislý podgraf. Každá zóna sa vytvára samostatne nezávislo na iných a k ich vytvoreniu je použitá modifikácia generátoru na procesný model. Rozdiel je v tom, že prvý uzol nie je štartovací a posledný uzol nie je koncový. Z toho plynú nasledujúce úpravy pri generovaní:

- v prípade pridávania cyklov je startIndexList v rozpätí od 1 do nodesCount - 1

  Dôvodom nato je fakt, že uzol s indexom 0 už nie je štartovací a teda z uzlu s indexom 1 vie ísť hrana do uzlu s indexom 0 a uzavrieť cyklus, dôvod na posun na nodesCount - 1 vyplýva z faktu, že posledný uzol už nie je koncový a teda z neho vie vychádzať hrana a taktiež uzavrieť cyklus.

- fáza vytvárania koncových uzlov sa úplne vynecháva z dôvodu, že koncové uzly vôbec nie sú na vstupe pre jednotlivé zóny

Poznamenajme, že takto vytvorené zóny ešte nespĺňajú vstup pretože neobsahujú InNodes/OutNodes ani nemajú ohodnotené hrany ohodnotením, ktoré by symbolizovalo výpadok pravdepodobnosti.

Zlúčenie úsekov a zón

V tejto fázy sa všetky úseky a všetky zóny spoja do jedného poľa. Poradie v poli je nasledujúce:

$$úsek_1, ZOP_1, úsek_2, ZOP_2, úsek_3, .., úsek_{zonesCount}, ZOP_{zonesCount}, úsek_{zonesCount+1}$$

Jednotlivé uzly sa preindexujú a nastavia sa im indexy odpovedajúce pozícii v poli.
Ako je vidieť toto pole uzlov ešte stále nepredstavuje súvislý graf a taktiež zóny ešte nemajú požadované InNodes a OutNodes. Zatiaľ len jednotlivé úseky a zóny sú súvislé podgrafy. Pre každú zónu sa InNodes vytvoria tak, že sa pridajú hrany, ktoré začínajú v nejakom uzly v úseku pred zónou a končia v nejakom uzly v danej zóne. Vybratie konkrétnych uzlov je náhodne avšak minimálne jeden z InNodes sa vytvorí tak, že sa vezme posledný uzol

z úseku pred zónou a potiahne sa z neho hrana do prvého uzlu v zóne, podobne to platí pre výstupné uzly kedy sa zo zóny ťahajú hrany do úseku za ňou s tou podmienkou, že aspoň jeden z OutNodes sa vytvorí tak, že sa vezme posledný uzol zo zóny a potiahne sa z neho hrana do prvého uzlu nasledujúceho úseku. Takýto prístup nielen zabezpečí potrebné InNodes a OutNodes zóny ale taktiež úseky so zónami prepojí. Obdobne to platí pre ďalšie zóny, vždy sa InNodes tvoria pomocou pridania hrán z predošlého úseku do zóny a OutNodes pomocou pridania hrán zo zóny do nasledujúceho úseku. Podmienka, že hrana musí viesť z posledného uzlu úseku do prvého uzlu zóny zabezpečuje, že v úseku nemáme nechcený koncový uzol a v zóne zas nemáme nechcený štartovací uzol. Reverzne to platí pri podmienke nato že aspoň jeden z OutNodes sa vytvorí tak, že sa pridá hrana vychádzajúca z posledného uzlu zóny do prvého uzlu nasledujúceho úseku. V tomto momente sú zóny skoro hotové, poslednou vecou ktorá zostáva je priradiť všetkým hranám zón ohodnotenie väčšie ako limit, ktoré predstavuje pravdepodobnosť výpadku pripojenia. Pre účely generovania bola zvolená konštanta 0,5.

Pridanie cyklov

V prípade pridávania cyklov sa jedná o skoro totožný postup ako v prípade pridávania cyklov v 1. Jediný rozdiel je v tom, že ako prvá vec sa musí vybrať úsek v ktorom sa cyklus vytvorí a v tomto úseku sa potom vytvára nová hrana, ktorá cyklus uzavrie. Teda vedie z uzlu s väčším indexom do uzlu s menším indexom. Oba uzly avšak musia byť v rovnakom úseku aby sme nechtiac nevytvorili nechcené cykly. Vzhľadom nato, že veľkosti úsekov môžu byť rôzne tak nejde použiť fixnú veľkosť cyklu ako v predošlom prípade, maximálna veľkosť cyklu sa určí ako počet uzlov úseku - cycleSizeDecrement, v prípade, ze by tento rozdiel bol menší ako 1 tak je maximum rovné 1.

Koncové uzly

Vytváranie koncových uzlov je opäť podobne ako v prípade vytvárania uzlov v 1. A rovnako ako v prípade pridávania cyklov je rozdiel v tom, že sa najprv musí vybrať úsek v ktorom sa koncový uzol vytvorí. Nová hrana kde sa hrany vedúce z nového konca napoja taktiež musí byť v rovnakom úseku.

Pridanie hrán

Pridanie zostávajúcich hrán funguje trochu inak ako predošlé prípady. Znovu sa jedná o obdobu pridávania hrán v procesnom modely ale tentokrát sa vyberá štartovací úsek a koncový úsek, kde platí, že koncový úsek môže byť rovnaký ako štartovací alebo môže byť viac napravo v poli, teda mat väčší index. V prípade, že je rovnaký tak pridanie hrany odpovedá postupu z generovania procesného modelu, v opačnom prípade koncový uzol môže mať ľubovoľný index v koncovom úseku, toto je z dôvodu inicializácie modelu a faktu, že keď koncový úsek má väčší index potom aj všetky jeho uzly majú

väčšie indexy ako vybraný štartovací uzol a teda máme istotu, že neuzavrieme cyklus.

Konečnosť

O konečnosť a úspech tohto postupu sa rovnako ako v predošlých prípadoch stará vhodne obmedzení vstup, postupne znižovanie veľkosti cyklu len na jednotkové cykly a nekonečný *while* cyklus, ktorý zabezpečuje, že pre krajné hodnoty medzí vstupu raz dôjde k situácii ilustrovanej na 5.1.3 teda, že sa podarí rozumne rozdistribuovať ako koncové uzly tak jednotkové cykly a pridať všetky hrany.

Algorithm 2 Generátor procesného modelu s ZOP

Input Zvalidovaný vstup od užívateľa.
Output Procesný model s ZOP odpovedajúci zadanému vstupu.

Vytvorenie úsekov.
Vytvorenie zón.
nodesArray ←Zlúčenie úsekov a zón.
iteration ← *0*
k ← *0*
cycleSizeDecrement ← *0*
while true **do**
    iteration ← *iteration* + *1*
    if *k* > *0* **then**
        Vrať *nodesArray* do pôvodného stavu.
        cycleSizeDecrement ← *cycleSizeDecrement* + *1*
    k ← *k* + *1*
    Pridanie cyklov. ▷ modifikuje *nodesArray*
    Oprava identifikátorov cyklov. ▷ modifikuje *nodesArray*
    Vytvorenie koncových uzlov. ▷ modifikuje *nodesArray*
    if Vytvorenie koncových uzlov nebolo úspešné. **then**
        Pokračuj ďalšou iteráciou while.
    Pridanie zostávajúcich hrán. ▷ modifikuje *nodesArray*
    if pridanie zostavajúcic hrán nebolo úspešné **then**
        Pokračuj ďalšou iteráciou while.
    return *nodesArray*

5.4 Rozšírenie procesného modelu

Pri rozšírení procesného modelu sú poskytované dve možnosti tou prvou je zhustiť model, teda pridať do už existujúceho modelu nové hrany a cykly. Tou druhou je jednoducho model zväčšiť, teda okrem nových hrán pridať aj uzly či už obyčajné alebo aj koncové. Generátor sa rozhoduje na základe formy vstupu, v obidvoch predošlých situáciach je vstupom počet uzlov, hrán, cyklov a koncových uzlov. Tie postupne označme ako nodesCount, edgesCount,

cyclesCount a endNodesCount. Tohto označenia sa budeme držať vo zvyšku tejto sekcie.

5.4.1 Zhustenie modelu

Pri zhustení modelu sa pridávajú len nové hrany a cykly. Vzhľadom nato, že sa v pôvodnom modeli nič neodoberá pridanie cyklu znamená pridanie hrany ktorá cyklus uzatvorí. Z vyššie spomenutého plynú nasledujúce obmedzenia na vstup:

1. Hodnoty pre nodesCount a endNodesCount musia byť rovné 0.
2. Minimálna hodnota pre cyclesCount je rovná 0 a maximálna hodnota je rovná 1000.
3. Minimálna hodnota pre edgesCount je rovná 1 a zároveň platí, že edgesCount nesmie byť menšie ako cyclesCount. Maximálna hodnota pre edgesCount je rovná 1000.

Popis postupu

V predošlých sekciách sme mali výhodu, že sme si mohli stavať model od začiatku a ďalej ho modifikovať s istotou že zachováme nami chcené vlastnosti. V tomto prípade tomu tak nie je. Umožňujeme načítať ľubovoľný model, stačí aby spĺňal požiadavky z 2.3. Tu vznikajú nasledujúce problémy:

- Pridanie hrany bez toho aby sme uzavreli cyklus.
- Pridanie presne jedného cyklu.

K tomuto bol navrhnutý postup ilustrovaný pseudokódom 3.

Algorithm 3 Generátor procesného modelu s ZOP

Input Zvalidovaný vstup od užívateľa a pôvodný model(*nodesArray*).
Output Rozšírený model v prípade úspechu inak výnimka.

```
1:  Nastavenia silne súvislých komponent.          ▷ Modifikuje nodesArray.
2:  Nastavenie zakázaných a povolených komponent.  ▷ Modifikuje
    nodesArray.
3:  Pridanie hrán.                                 ▷ Modifikuje nodesArray.
4:  if Pridanie hrán nebolo úspešné. then
5:      Vyhodenie výnimky a ukončenie algoritmu.
6:  Detekcia a nastavenie zakázaných a povolených komponent pre uzly.
7:  Pridanie cyklov.                               ▷ Modifikuje nodesArray.
8:  if Pridanie cyklov nebolo úspešné. then
9:      Vyhodenie výnimky a ukončenie algoritmu.
10: return nodesArray
```

Nastavenie silne súvislých komponent

Tarjanov algoritmus je algoritmus na nájdenie silne súvislých komponent v orientovanom grafe [37]. Je založený na prehľadavaní do hĺbky. Keď sa pri prehľadavaní spracováva uzol nastaví sa mu *id* a tzv. *link* hodnota. Súčasný uzol sa následne označí ako navštívený a umiestni sa na zásobník. Pri návrate sa kontroluje predošlý uzol, v prípade, že je na zásobníku tak sa súčasnému uzlu nastaví ako *link* hodnota minimálna hodnota *link* z súčasného uzlu a predošlého uzlu. Až sú navštívení všetci susedia a súčasný uzol bol začiatok silne súvislej komponenty tak sa komponenta označí. To že uzol bol začiatok pre komponentu sa zistí porovnaním *id* a *link* hodnoty. Pre náš prípad bol Tarjanov algoritmus mierne modifikovaný, v časti kde sa nastavuje *id* komponenty pre jednotlivé uzly zo zásobníka sa zásobník najprv preiteruje s cieľom zistiť počet všetkých uzlov danej komponenty. V kontexte tejto práce pod pojmom veľkosť komponenty myslíme počet uzlov danej komponenty. Táto modifikácia je na priloženom obrázku5.3 kódu. Riadok 397 je vyššie spomenutá kontrola z Tarjanovho algoritmu, ktorá detekuje či má označovať komponentu. Našej modifikácii odpovedajú riadky 400-405, ktoré sa starajú o spočítanie veľkosti komponenty a ďalej riadok 411, ktorý jednotlivým uzlom pridá informáciu o veľkosti komponenty. S touto informáciou budeme ďalej pracovať v ďalších častiach postupu.

```
397         if (tarjanNode.getLink() == tarjanNode.getCid()) {
398             ExtendedProcessNode next;
399
400             int size = 0;
401             Iterator<ExtendedProcessNode> iterator = stack.iterator();
402             do {
403                 next = iterator.next();
404                 size++;
405             } while (next.getId() != tarjanNode.getId());
406
407             do {
408                 next = stack.pop();
409                 next.setCid(componentId);
410                 next.getDeniedCids().add(componentId);
411                 next.setComponentSize(size);
412             } while (next.getId() != tarjanNode.getId());
413             componentId++;
414         }
```

Obrázok 5.3: Ukážka modifikácie Tarjanovho algoritmu.

Nastavenie zakázaných a povolených komponent

Na obrázku 5.4 je znázornený graf, ktorý by mohol predstavovať model IoT systému, ktorý chceme rozšíriť. Vidíme, že ak by sme chceli pridať presne

jednu hranu, ktorá nemá uzavrieť cyklus nemôžme ju viesť napríklad z uzlu D do uzlu B. Potrebujeme vedieť, že vzhľadom na štartovací uzol je uzol B bližšie než uzol D a z uzlu B vedie hrana do uzlu D. Pre tento prípad by stačilo zakázať hrany do rodičov. Taktiež platí, že hrana z D do A by uzavrela cyklus a A nie je priamy predok D. Z toho vyplýva, že pre uzol D potrebujeme informáciu o jeho predkoch aby sme nechtiac neuzavreli cyklus. Je znateľné, že vzhľadom nato, že uzly A,B,C,H tvoria jednu silne súvislú komponentu nemôže viesť hrana z D ani do C alebo H. Riešením by mohlo byť pre uzly držať množinu zakázaných identifikátorov, tieto identifikátory by reprezentovali uzly, ktoré sú v grafe pred daným uzlom. To by išlo implementovať tak, že by sme z každého uzlu postupne prešli všetkých jeho potomkov a ich potomkov apod. Avšak toto nie je nutné, príklad vyššie ilustruje, že nám stačí vedieť o danej komponente. Obecne pokiaľ z nejakého uzlu v vedie hrana do uzlu u tak potom pridanie hrany z u do ľubovoľného uzlu, ktorý je v rovnakej silne súvislej komponentu ako v by vytvorilo minimálne jeden cyklus. Na druhej strane nemôžme povedať, že nám stačí každý uzol navštíviť raz, napríklad ak by sme začali v START pokračovali A potom B a potom D a uzlu D nastavili informáciu o komponente v ktorej je B tak by to síce riešilo problém, žeby sme vedeli ze z uzlu D nemôžme viesť hranu do A,C,B,H. Avšak ako ilustruje graf na obrázku 5.4 z uzlu D nemôžme viesť hranu ani do E, F, G to by taktiež uzavrelo cyklus, teda nejde jednoducho označiť D ako navštívený a už ho ďalej nespracovávať, zároveň potrebujeme niekedy skončiť, zaručiť že sa vrámci komponenty nebudeme točiť donekonečna. K tomu bol navrhnutý nasledujúci postup inšpirovaný prehľadávaním do hľbky 5.5.

Ilustrujme tento prechod na vyššie zvolenom príklade z 5.4.

1. currentNode = START, next = E
   E je v komponente o veľkosti 1, *if* na riadku 126 sa preskočí a E sa vloží na zásobník.

2. currentNode = START, next = A
   A je v komponente o väčšej veľkosti než 1 a zároveň sa identifikátory ich komponent nerovnajú, A dostane *mark* rovné identifikátoru currentNode, v tomto prípade nech je to 0.

3. currentNode = A, next = H
   Identifikátory komponent sa rovnajú a zároveň veľkosť komponenty je väčšia ako 1, prebehne riadok 133, uzol H má *mark* rovný *marku* A to je 0, H sa vloží na zásobník

4. currentNode = H, next = I
   I je v komponente o veľkosti 1, *if* na riadku 126 sa preskočí a I sa vloží na zásobník

5. currentNode = H, next = B
   B je v rovnakej komponente ako H, prebehne riadok 133 a B sa vloží na zásobník.

6. currentNode = B, next = D
   D je v komponente o veľkosti 1, *if* na riadku 126 sa preskočí a D sa vloží na zásobník.

7. currentNode = D, next = END_1
   END_1 sa vloží na zásobník.

8. Spracuje sa END_1 ale z neho nejdú hrany.

9. currentNode = B, next = C
   C je v rovnakej komponente ako B prebehne riadok 133.

10. currentNode = C, next = A
    A je v komponente väčšej ako 1 a zároveň sa rovnajú identifikátory komponent a *mark*, A má *mark* z bodu 2 a tento *mark* sa postupne cez H, B dostal do C v predošlom bode, prebehne riadok 135 a A sa znovu na zásobník nepridá, nie je v tejto chvíli potrebné potrebné všetky uzly z neho sme spracovali.

11. Raz dôjde k tomu, že z E sa pridá F, z F sa pridá G a nastane situácia kedy currentNode = G a next = C, v tomto prípade dostane C ako *mark* identifikátor G a celá komponenta sa spracuje znovu(vrátane potomkov) a to je chcená situácia predošlý bod nám zaručuje koniec postupu aj v prípade netriviálnych komponent, no *mark* a tento prístup nám zaručuje, že uzly C, H , I, END_2, B, D, END1 sa dozvedia o tom, že z nich nesmie ísť hrana do G, E, F. Podobne ako je tomu v predošlom bode až sa do C znovu dostaneme z uzlu B tak sa C nepridá pretože B už bude mat rovnaký *mark*.

Jednotlivé body vyššie ilustrovali ako funguje prechod grafom, riadok na obrázku 5.5 125 sa stará o zväčšenie množiny zakázaných komponent pre jednotlivé uzly. To funguje tak, že pri inicializácii silne súvislých komponent, keď sa daná komponenta označuje, pridáme uzlu do množiny zakazáných komponent identifikátor komponenty ktorej je súčasťou. To zabezpečí riadok 410 na obrázku 5.3. Pri prehľadávaní keď sa prechádzajú potomkovia uzlu sa množina aktuálneho potomka zväčší o zakázané komponenty rodiča.

Pre jednotlivé uzly nastavenie povolených komponent funguje tak, že sa vezme množina všetkých komponent a odmaže sa podmnožina odpovedajúca zakázaným komponentám.

Pridanie hrán

Tato časť ilustruje ako funguje pridanie hrán.

1. Nainicializuje sa zoznam prirodzených čísel od 0 po nodesCount-1, označme ako startIndexList. Poznamenajme, že tieto čísla odpovedajú identifikátorom jednotlivých uzlov načítaného modelu.

2. Vyberie sa zo zozamu náhodne jeden prvok, označme ako startIndex, v prípade, že startIndex je odpovedá uzlu, ktorý je koncový alebo uzlu, ktorý je vo vnútri zóny tak sa startIndex odmaže zo zoznamu startIndexList a výber sa opakuje.

3. Uzol so startIndex drží zoznam povolených komponent, ten sa roztriedi a vyberie sa prvá komponenta.

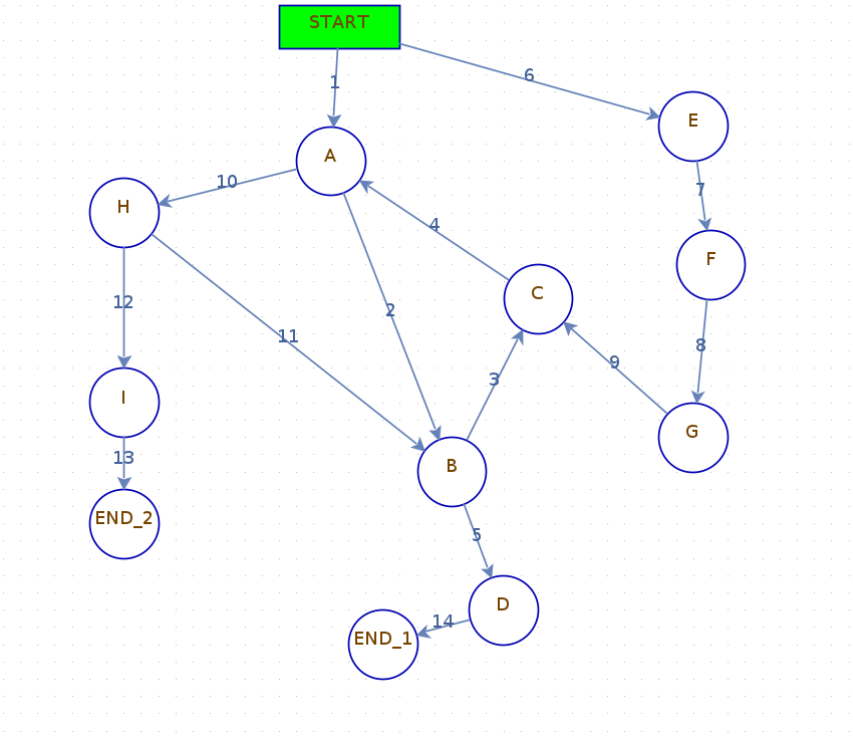
4. Z vybranej komponenty sa postupne iterujú jej uzly s cieľom nájsť vhodný koniec hrany, označme aktuálne vybraný prvok ako endIndex potom sa musí overiť nasledujúce:

   - startIndex odpovedá štartovaciemu uzlu a endIndex koncovému uzlu
   - hrana už existuje
   - uzol s endIndex je vo vnútri zóny

   V prípade, že niečo vyššie spomenuté je splnené endIndex nie je validný a pokračuje sa výberom ďalšieho prvku. V prípade, že sa nenájde vhodný kandidát, tak sa zas postupuje výberom ďalšej komponenty. V prípade, ze sa nenajde vhodný kandidát ani v zvyšných komponentách, tak sa startIndex odmaže z startIndexList a postup sa opakuje. V opačnom prípade sa pridá hrana vedúca z uzlu so startIndex do uzlu s endIndex a pokračuje sa bodom 2 kým niesu pridané potrebné hrany.

5. v prípade, že uz sú pridané všetky požadované hrany postup končí úspechom. V prípade, že tomu tak nie je a startIndexList je prázdny tak postup končí neúspešne.

Ako je vyššie vidieť tak ako počiatočný tak koncový uzol novo pridanej hrany sa kontroluje či nie je vo vnútri zóny, tým myslime, že sa nejedná ani o IN ani OUT uzol zóny. Táto kontrola je preto aby sme stávajúce zóny nenarušili a nevytvorili nové IN/OUT uzly.



Obrázok 5.4: Príklad modelu.

```
117     public void dfsSetDeniedComponents() {
118         var stack = new Stack<ExtendedProcessNode>();
119         var currentNode :ExtendedProcessNode = nodes[startIndex];
120         stack.push(currentNode);
121         while (!stack.isEmpty()) {
122             currentNode = stack.pop();
123             componentNodes[currentNode.getCid()].add(currentNode.getId());
124             for (var next : currentNode.getNextNodes()) {
125                 next.getDeniedCids().addAll(currentNode.getDeniedCids());
126                 if (next.getComponentSize() > 1) {
127                     if (currentNode.getCid() != next.getCid()) {
128                         if (currentNode.getCid() == next.getMark()) {
129                             continue;
130                         }
131                         next.setMark(currentNode.getCid());
132                     } else if (next.getMark() != currentNode.getMark()) {
133                         next.setMark(currentNode.getMark());
134                     } else {
135                         continue;
136                     }
137                 }
138                 stack.push(next);
139             }
140         }
141     }
```

Obrázok 5.5: Prehľadávanie grafu s cieľom určiť zakázané komponenty.

Pridanie cyklov

Tato čas ilustruje ako funguje pridanie cyklov.

1. Na začiatku sa vytvorí zoznam možných štartov, teda uzlov z ktorých hrana uzatvarajúca cyklus bude vychádzať, jednotlivé uzly musia spĺňať všetky nasledujúce vlastnosti:

   - uzol je v komponente o veľkosti 1
   - uzol nie je štartovací uzol modelu
   - uzol nie je koncový uzol modelu
   - uzol nie je vo vnútri zóny

   tento zoznam sa používa pre vytvorenie všetkých požadovaných cyklov, ďalej počítajme situáciu, že chceme pridať cyklus a zo zoznamu sme vybrali začiatok nového cyklu, označme ako startIndex.

2. Vieme, že náš startIndex odpovedá uzlu, ktorý nie je štartovací uzol modelu, z toho vyplýva, že doň nutne vedie aspoň jedna hrana, uzol so startIndex je pre minimálne jednu hranu koncový uzol hrany, vezmeme

množinu vstupných uzlov týchto hrán a označíme ju ako previousSet, v previousSet sa snažíme nájsť vhodného kandidáta, preňho musia platiť podmienky z bodu 1 a zároveň nesmie viesť hrana z uzlu s indexom startIndex do nášho kandidáta. Ďalej tento vybraný uzol má sám o sebe výstupné hrany, tieto hrany majú niekde koncové uzly, musí platiť, že žiaden z týchto koncových uzlov nie je v previousSet, v opačnom prípade takýto uzol sa vloží do množiny forbiddenNodes a vyberá sa nový kandidát.

3. Pokiaľ sa v predošlom bode našiel vhodný kandidát tak cyklus z daného startIndex vieme uzavrieť a to minimálne o veľkosti 1 ak nie startIndex sa hľada znovu a bod 2 sa opakuje, tento vhodný uzol dáme do zoznamu nodesInCycle.

4. Ďalej vyberáme posledný prvok v zozname nodesInNewCycle, označme ho ako *parent*.

5. Podobne ako v bode 2 vezmeme z *parent* množinu uzlov jeho vstupných hrán, označme ako previousSetOfParent, z tejto množiny sa snažíme nájsť vhodného kandidáta, označme ako *previous* ktorý spĺňa nasledujúce:

   a. Uzol je v komponente o veľkosti 1, nie je štartovací uzol modelu, uzol nie je vo vnútri zóny a nevedie doň hrana z uzlu so startIndex.

   b. Jednotlive uzly z forbiddenNodes v množine zakázaných komponent neobsahujú komponentu v ktorej je *previous*.

   c. Koncové uzly výstupných hrán z *previous* s výnimkou *parent* niesu v nodesInNewCycleList.

   d. Neexistuje priama hrana z *previous* do uzlu s startIndex.

   Pokiaľ sa takýto kandidát nájde tak sa pridá do nodesInNewCycle a postup sa opakuje, teda počet uzlov nového cyklu sa zväčší, raz nutne nastane situácia, že sa už vhodný kandidát nenájde kvôli podmienkam spomenutým vyššie.

6. Nový cyklus sa uzavrie pridaním hrany z uzlu so startIndex do posledného uzlu z nodesInNewCycle.

5.4.2 Zväčšenie modelu

Tento problém bol riešený tak, že pre vstup sa vygeneruje nový model a to postupom v 5.1. Tento model sa potom napojí na stávajúci model, ktorý chceme rozšíriť. Pri napojení sa použije jedna hrana vedúca z pôvodného modelu do nového modelu. Počiatočný uzol tejto hrany môže byť ľubovoľný uzol v pôvodnom modeli s výnimkou koncových uzlov a taktiež uzlov, ktoré sú vo vnútri zóny. Koncový uzol tejto hrany musí byt štartovací uzol nového modelu aby zostala vlastnosť jedného štartovacieho uzlu. Je povolená taktiež aj situácia, že v požiadavkách na rozšírenie bude endNodesCount rovné nule, potom sa pridá ešte jedna hrana a tá bude viesť z koncového uzlu nového

modelu do pôvodného modelu a to do ľubovolného uzlu, ktorý nie je v zóne.

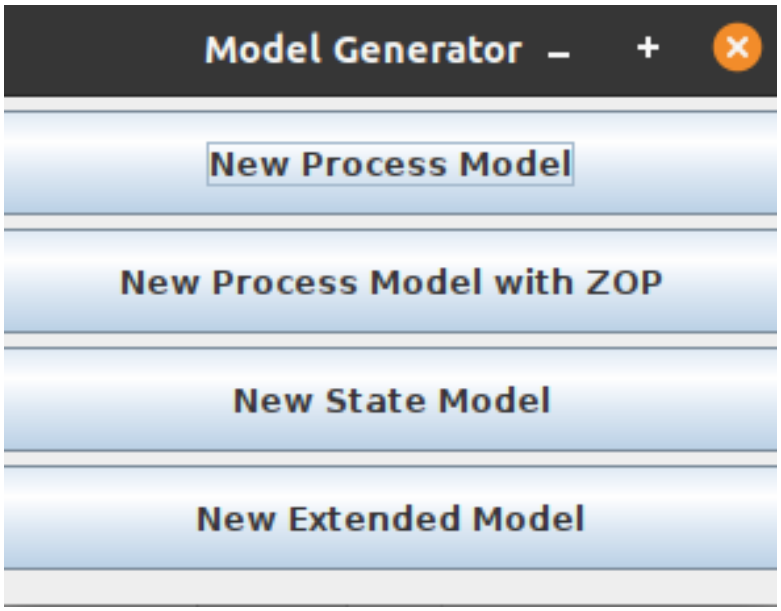
Nie je náhodou, že sme jednotlivé vstupy označili rovnako ako v sekcii 5.1. Požiadavky na vstup ostávajú ako v prípade odkaz 5.1.1. No vzhľadom na text vyššie sú dva rozdiely:

1. Hodnota pre minEdgesCount musí byť vačša o 1 v prípade, že endNodesCount je aspoň 1. Ak endNodesCount je rovné 0 tak minEdgesCount musí byť vačšie o 2.
   To vyplýva z potreby napojiť nový model na stávajúci model.

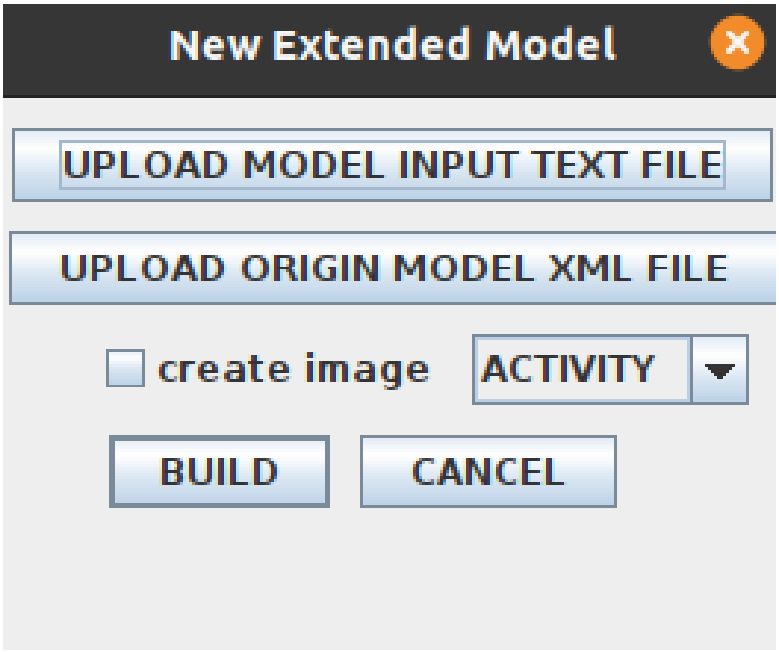
2. Hodnota pre endNodesCount môže byť rovná 0.

5.5 Užívateľské rozhranie

Táto sekcia ukazuje navrhnuté užívateľské rozhranie, čo odpovedá požiadavku FR1. Hlavná obrazovka aplikácie je na obrázku 5.6 a pozostáva z štyroch tlačidiel, ktoré odpovedajú funkcionalitám z požiadavok FR9, FR10, FR11 a FR12. Po kliknutí na ľubovoľné tlačidlo sa zobrazí dialóg v ktorom sa zadáva vstup a spúšťa generovanie modelu. Pre požiadavok FR11 je tento dialóg na obrázku 5.7. Pre ostatné možnosti sa jedná o obdobu rovnakého dialógu, kde sú možnosti zo vstupu len znížené podľa potreby.



Obrázok 5.6: Ukážka hlavnej obrazovky aplikácie.



Obrázok 5.7: Ukážka dialógu na rozšírenie procesného modelu.

Kapitola 6

Testovanie

Táto kapitola popisuje ako bola výsledná aplikácia otestovaná. Cieľom testovania bolo v prvom rade overiť, že výsledná aplikácia spĺňa jednotlivé funkčné požiadavky. Testovanie môžme rozdeliť na pozitívne a negatívne. Kde pozitívne testovanie predstavuje overenie, že pre vhodné vstupné dáta dostaneme očakávaný výstup. Negatívne testovanie sa týka nevalidných vstupov, cieľom je hlavne overiť, že aplikácia nespadne. V prípade nevhodného vstupu by užívateľ mal dostať aj vhodnú spätnú väzbu, to odpovedá nášmu požiadavku FR6. Splnenie jednotlivých funkčných požiadavok bolo overené pomocou užívateľského testovania, kde boli prejdené pozitívne aj negatívne testovacie scenáre. Pre najdôležitejšie časti kódu boli napísane aj automatizované testy. Na konci tejto kapitoly je taktiež ukážka, koľko času vygenerovanie modelu zaberie pre nami vybrané vstupy.

6.1 Užívateľské testovanie

Táto sekcia ukazuje ako bola aplikácia manuálne otestovaná. Nižšie je ilustrovaný ako pozitívny tak aj negatívny prechod.

Pozitívne testovanie

V tejto časti bola otestovaná funkčnosť nasledujúcich požiadavok: FR1, FR2, FR3, FR4, FR5, FR7, FR8, FR9, FR10, FR11, FR12, FR13, FR15. Body nižšie ilustrujú ako k tomu bolo pristúpené.

1. Spustenie aplikácie.
2. Výber jednej z 4 možností generovania podľa požiadavok FR9, FR10, FR11 a FR12.
3. Zadanie validného vstupu podľa požiadavok a obmedzení na vstup.
4. Spustenie generovania.
5. Manuálna kontrola výsledného súboru. Výsledný súbor musí obsahovať: súbor s logovaním generovania, jednotlivé chcené modely a v prípade možnosti *vykresliť obrázok* aj obrázky modelov vo formáte *jpg*.

6. Načítanie vygenerovaných modelov do aplikácie Oxygen.

Negatívne testovanie

V tejto časti boli overené požiadavky FR6 a FR14. Prístup je ilustrovaný v jednotlivých bodoch nižšie.

1. Spustenie aplikácie.
2. Výber jednej z 4 možností generovania podľa požiadavok FR9, FR10, FR11 a FR12.
3. Zadanie nevalidného vstupu. Overenie nasledujúcich možností:
   - Vstupný textový súbor neobsahuje všetky potrebné hodnoty.
   - Vstupný textový súbor obsahuje všetky hodnoty ale aspoň jedna z nich je mimo povolený rozsah.
   - Špeciálne pre prípad FR11, vstup vo forme kombinácie modelu a požadovaného rozšírenia, ktoré sa nemôže podariť.
4. Spustenie generovania.
5. Kontrola na existenciu a zmysluplnosť spätnej väzby.

6.2 Automatizované testovanie

Pre automatizované testovanie boli zvolené jednotkové testy. Jednotkový test je časť kódu, ktorá overuje funkčnosť nejakej izolovanej časti aplikácie, napríklad jednej metódy. K implementácii jednotkových testov bola využitá knižnica *JUnit* [38].

Z pohľadu kódu bola jednotkovými testami otestovaná funkčnosť validovania užívateľského vstupu a samotného generovania modelov. Pri generovaní modelu z požiadavkov FR9 a FR11 plynie potreba na overenie, že výsledný model má požadovaný počet hrán, počet uzlov, počet cyklov a počet koncových uzlov. Požiadavok FR10, k tomu pridáva ešte overenie počtu zón a ich IN/OUT uzlov. Požiadavok FR12 zas pridáva overenie počtu počiatočných stavov a testStart, testEnd, testStartEnd atribútov. Výsledný model obsahuje dostatok údajov aby toto išlo jednoducho overiť s výnimkou overenia výsledného počtu cyklov. K overeniu, že výsledný model má presne požadovaný počet cyklov bol využitý algoritmus od D. B. Johnsona: *FINDING ALL THE ELEMENTARY CIRCUITS OF A DIRECTED GRAPH** [39]. Tento algoritmus nebol nijak modifikovaný a konkrétna implementácia bola prevzatá z [40].

6.3 Výsledky behu

Ako je vidieť z požiadavok a popisu generovania jednotlivých modelov, tak vstup pre generovanie obsahuje viacero hodnôt, taktiež bolo vytvorené pomerne veľké rozpätie na vstupné údaje. Ako ukazujú jednotlivé postupy generovania, tak časové trvanie behu silne závisí na vstupe. Sekcia venovaná generovaniu procesného modelu, ukazuje, že v prípade vysokého počtu cyklov a zároveň celkového počtu hrán alebo koncových uzlov sa generovanie nemusí podariť na prvý krát ale musí sa pristúpiť k znižovaniu maximálnej veľkosti cyklu. Ďalší problém je náhodný výber v prípade vytvárania cyklov, koncových uzlov alebo pridávania zostávajúcich hrán. Všetky tieto veci silne komplikujú určenie asymptotickej časovej náročnosti a z pohľadu tejto práce a predpokladanej veľkosti vstupu nie je dôležitá. Bez ohľadu nato by bolo dobré vedieť ako dlho približne vytvorenie finálneho modelu trvá. Ako hovorí predošlá kapitola, tak jednotlivé možnosti generovanie sú založené na generovaní procesného modelu a preto práve táto časť bola hlbšie skúmaná. Vzhľadom na rešerš a obmedzenia na vstup boli stanovené nasledujúce testovacie instancie vstupu:

Názov	nodesCount	edgesCount	endNodesCount	cyclesCount
vstup 1	20	40	2	4
vstup 2	200	400	20	40
vstup 3	500	1000	100	100

Tabuľka 6.1: Testovacie vstupné údaje.

Generovanie procesného modelu bolo opakovane spustené pre tieto vstupy s cieľom zistiť výsledné trvanie práce. Výsledky sú v tabuľke nižšie 6.2. Ako je vidieť pre hodnoty podobné tým z rešerše je požiadavok NR3 s prehľadom splnený, zhoršuje sa to až v prípade, že počet koncových uzlov a/alebo cyklov sa viac blíži celkovému počtu uzlov.

Vstup	Beh	Čas [ms]
vstup 1	1	30
vstup 1	2	27
vstup 1	3	12
vstup 1	4	16
vstup 1	5	22
vstup 2	1	2175
vstup 2	2	560
vstup 2	3	638
vstup 2	4	191
vstup 2	5	791
vstup 3	1	35323
vstup 3	2	38052
vstup 3	3	37820
vstup 3	4	29953
vstup 3	5	31044

Tabuľka 6.2: Výsledky behu generátoru.

Kapitola 7

Záver

Cieľom práce bolo vykonať rešerš IoT systémov a upresniť požiadavky na generátor modelov určených k testovaniu algoritmov na generovanie testovacích scenárov z modelu. Potrebnou súčasťou bolo taktiež zoznámiť sa s aplikáciou Oxygen. Ďalej tento generátor navrhnúť, implementovať a otestovať. Práca obsahuje popis potrebných pojmov z teórie grafov, nasleduje popis IoT systémov a zoznámenie čitateľa s problematikou testovania IoT systémov v prípade výpadku alebo obmedzeného sieťového pripojenia. Text pokračuje špecifikovaním požadovaného modelu a to ako procesného tak stavového. Nasleduje požadovaná rešerš a jej výsledky.

Počas riešenia problému boli formálne spísané funkčné a kvalitatívne požiadavky na výsledný produkt a to vrátane ich priorít. Text nasleduje popisom použitých technológií a rozborom navrhnutej architektúry. Z požiadavok boli splnené všetky požadované situácie a to vygenerovať procesný model, procesný model so zónamy, stavový model a rozšíriť už existujúci procesný model. Popis ako tieto funkcionality boli dosiahnuté nasleduje hneď po popise architektúry.

Výsledkom práce je Java desktopová aplikácia, ktorá umožňuje užívateľovi načítať vstup podľa požiadavok a vygenerovať alebo rozšíriť model. Tento model ide taktiež vizualizovať vo formáte *jpg*. Jednotlivé funkčné požiadavky boli manuálne otestované a pre dôležité časti kódu nechýbajú ani jednotkové testy. Výsledné modely ide úspešne načítať do aplikácie Oxygen a môžu byť použité pri testovaní existujúcich alebo novo vyvíjaných algoritmov na generovanie testovacích scenárov z modelu IoT systému. Z pohľadu oficiálneho zadania boli splnené všetky požiadavky, oproti zadaniu bolo ešte po diskusii so školiteľom pridané: vykreslenie modelu, užívateľské rozhranie[1] a možná budúca integrácia s Oxygenom. Výsledná práca ešte nebola integrovnaná do Oxygenu, no bol použitý rovnaký programovací jazyk a verzia[2] aby to bolo bez problémov možné v budúcnosti vykonať.

Práca úzko súvisí s aplikáciou Oxygen a to ako čo sa týka načítania vstupného modelu tak aj forma výstupu je podľa schémy definovanej Oxygenom.

[1] bez neho by sa mohlo jednať len o konzolovú aplikáciu
[2] Java SE 8

Preto ako nasledujúca práca dáva zmysel zakomponovať tento softvér ako jednu z častí Oxygenu. Ďalšou možnosťou budúceho vývoja by bolo viac upresniť vstupné údaje, teda požadované vlastnosti výsledného modelu a to napríklad v podobe obmedzenia na maximálny počet výstupných hrán z jedného uzlu.

Literatúra

[1] Jiří Demel: *Grafy a jejich aplikace*, rok vydania 2002

[2] Kevin Ashton: *That 'Internet of Things' Thing* dostupné na `https://www.rfidjournal.com/that-internet-of-things-thing` [vid. 4.5.2021].

[3] Madakam, S., Lake, V., Lake, V., & Lake, V. Internet of Things (IoT): A literature review. Journal of Computer and Communications, 3(05), 2015.

[4] Nunberg, G. (2012) The Advent of the Internet: 12th April, Courses.

[5] Tsiatsis, Vlasios, et al. Internet of Things : Technologies and Applications for a New Age of Intelligence. London: Academic Press, 2019.

[6] Článok: *How Many IoT Devices Are There in 2021? [All You Need To Know]* dostupné na `https://techjury.net/blog/how-many-iot-devices-are-there/#gref` [vid. 4.5.2021].

[7] Článok: *Brent Heslop By 2030, Each Person Will Own 15 Connected Devices — Here's What That Means for Your Business and Content* dostupné na `https://www.martechadvisor.com/articles/iot/by-2030-each-person-will-own-15-connected-devices-heres-what\-that-means-for-your-business-and-content/` [vid. 4.5.2021].

[8] Dave Evans (2011) *The Internet of Things How the Next Evolution of the Internet Is Changing Everything* dostupné na `https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf` [vid. 4.5.2021].

[9] M. R. Palattella, M. Dohler, A. Grieco, G. Rizzo, J. Torsner, T. Engel, and L. Ladid, *Internet of things in the 5g era: Enablers, architecture, and business models* IEEE Journal on Selected Areas in Communications, vol. 34, no. 3, pp. 510–527, March 2016.

[10] M. Woolley *Bluetooth 5: Go faster. go further | bluetooth technology website.* dostupné na `https://www.bluetooth.com/bluetooth-resources/bluetooth-5-go-faster-go-further/`

[11] online článok: *10 INTERNET OF THINGS (IOT) HEALTHCARE EXAMPLES, AND WHY THEIR SECURITY MATTERS* dostupné online na `https://ordr.net/article/iot-healthcare-examples/` [vid. 4.5.2021].

[12] Miroslav Bures, Xavier Bellekens, Karel Frajtak, Bestoun S. Ahmed *A Comprehensive View on Quality Characteristics of the IoT Solutions*

[13] online zdroj TMAP *Quality Characterstics in an IoT environment* dostupné na `https://www.tmap.net/wiki/quality-characterstics-iot-environment` [vid. 4.5.2021].

[14] Matej Klima, Miroslav Bures *Testing Tool for IoT Systems Operating with Limited Network Connectivity*

[15] online článok *Model-Based Testing (MBT) Overview* dostupné na `https://www.broadcom.com/info/continuous-testing/model-based-testing` [vid. 4.5.2021].

[16] Vedecká databáza *ScienceDirect* dostupné online na `https://www.sciencedirect.com/`

[17] Vedecká databáza *IEEE Xplore* dostupné online na `https://ieeexplore.ieee.org/Xplore/home.jsp`

[18] Courage Kpotosu and Scholastica Memusi *Smart Agriculture Using IoT* dostupné na `https://www.researchgate.net/publication/335971122_Smart_Agriculture_Using_IoT`

[19] Md. SanwarHossaina, MostafizurRahmana, Md. TuhinSarkerb, Md. ErshadulHaquecAbuJahida *A smart IoT based system for monitoring and controlling the sub-station equipment* dostupné na `https://www.sciencedirect.com/science/article/pii/S2542660518301628?via%3Dihub`

[20] Amgad MuneerSuliman, Mohamed FatiSuliman Mohamed Fati, Saddam Fuddah *Smart health monitoring system using IoT based smart fitness mirror* dostupné na `https://www.researchgate.net/publication/338838854_Smart_health_monitoring_system_using_IoT_based_smart_fitness_mirror`

[21] G K Jakir Hussain, O S Dharshini, G U Kavipriya, G Jeevanandhini *E-Parking Reservation system based on IoT for smart cities* dostupné na `https://www.researchgate.net/publication/333561525_E-Parking_Reservation_system_based_on_IoT_for_smart_cities`

[22] Henry Fuentes, David Mauricio *Smart water consumption measurement system for houses using IoT and cloud computing* dostupné na `https://link.springer.com/article/10.1007/s10661-020-08535-4`

[23] Alok Kumar Gupta, Rahul Johari *IOT based Electrical Device Surveillance and Control System* dostupné na https://ieeexplore.ieee.org/document/8777342

[24] Ch Rajendra Prasad *Patient Health Monitoring using IoT* dostupné na https://www.researchgate.net/publication/331383569_Patient_Health_Monitoring_using_IoT

[25] Gede SusramaNi, Luh Wiwik Sri R.G, Slamet Winardi, Tri Andjarwati *Progressive Parking Smart System in Surabaya's Open Area Based on IoT* dostupné na https://www.researchgate.net/publication/343188125_Progressive_Parking_Smart_System_in_Surabaya%27s_Open_Area_Based_on_IoT

[26] Vignesh Aravindan, Darshan James *Smart Homes using Internet of Things* dostupné na https://www.scribd.com/document/427802752/smart-homes-uisng-internet-of-things-pdf

[27] Morteza Hadipour, Javad Farrokhi Derakhshandeh, Mohsen Aghazadeh Shiran, Reza Rezaei *Automatic washing system of LED street lighting via Internet of Things* dostupné na https://www.sciencedirect.com/science/article/pii/S2542660518300465?via%3Dihub

[28] E.B.Priyanka, C.Maheswari, S.Thangavel *IoT based field parameters monitoring and control in press shop assembly* dostupné na https://www.sciencedirect.com/science/article/pii/S254266051830060X?via%3Dihub

[29] HATTON, S. *Choosing the right prioritisation method* v 19th Australian Conference on Software Engineering (aswec 2008)

[30] NEUSTADT, I. ARLOW, J. *UML 2 a unifikovaný proces vývoje aplikací* Computer Press, Albatros Media as, 2016. ISBN 978-802511503.

[31] Stránka frameworku JUNG dostupné na http://jung.sourceforge.net/ [vid. 8.6.2021].

[32] *Java Swing* online zdroj Techopedia, dostupné na https://www.techopedia.com/definition/26102/java-swing

[33] *XML (Extensible Markup Language)* dostupné na https://whatis.techtarget.com/definition/XML-Extensible-Markup-Language [vid. 11.6.2021].

[34] *Welcome to Apache Maven* dostupné na https://maven.apache.org/index.html

[35] *Apache Commons IO* dostupné na https://commons.apache.org/proper/commons-io/

[36] Philippe Kruchten *The 4+1 View Model of Architecture*, dostupné na https://www.researchgate.net/publication/220018231_The_41_View_Model_of_Architecture

[37] Abhishek Dey *Tarjan's Algorithm for finding Strongly Connected Components in Directed Graph*, dostupné na https://www.thealgorists.com/Algo/GraphTheory/Tarjan/SCC [vid. 13.6.2021].

[38] JUnit dostupné na https://junit.org/junit5/

[39] DONALD B. JOHNSON: *FINDING ALL THE ELEMENTARY CIRCUITS OF A DIRECTED GRAPH** dostupné na https://www.cs.tufts.edu/comp/150GA/homeworks/hw1/Johnson%2075.PDF

[40] Tushar Roy *Implementácia Johnsnovho algoritmu na nájdene všetkých cyklov v grafe* dostupné na https://github.com/mission-peace/interview/blob/master/src/com/interview/graph/AllCyclesInDirectedGraphJohnson.java [vid. 4.4.2021].

Dodatok A

Zoznam skratiek

IoT - Internet of Things

UML - Unified Modeling Language

JVM - Java Virtual Machine

XML - Extensible Markup Language

JUNG - Java Universal Network/Graph Framework

Dodatok B

Obsah priloženého CD

aplikacia

Zdrojové kódy.

Software_IoT_Model.pdf

Text práce.

vstupPriklady

Priečinok s príkladmy vstupov.

Generator-1.0-SNAPSHOT-jar-with-dependencies.jar

Spustiteľná aplikácia.

Dodatok C
Užívateľská príručka

Aplikácia je *Generator-1.0-SNAPSHOT-jar-with-dependencies.jar*(musí byť nastavené *Allow executing file as program*) k jej spusteniu je potrebná podpora Java SE8. Po spustení sa objaví hlavná obrazovka ako je na obrázku 5.6. Jednotlivé tlačidlá otvárajú dialóg na zadanie požadovaného vstupu a spustenie generovania požadovaného modelu. Užívateľské rozhranie by malo byť samovysvetľujúce s výnimkou formy vstupného textového súboru. Toto riešia nasledujúce sekcie.

C.1 Procesný model

Vstupný súbor

Očakávaný vstupný textový súbor má nasledujúci formát. Prvý riadok obsahue hodnotu koľko modelov chceme vygenerovať, označme ako N. Nasleduje N riadkov, kde každý riadok obsahuje vstup pre daný model vo formáte: nodesCount edgesCount endNodesCount cyclesCount. Očakávané sú prirodzené čísla podľa obmedzení z 5.1.1 oddelené medzeramy.

C.2 Procesný model so ZOP

Vstupný súbor

V tejto časti budeme používať pojmy ako sú vysvetlené v 5.3.1. Očakávaný vstupný textový súbor má nasledujúci formát. Prvý riadok obsahuje hodnotu koľko modelov chceme vygenerovať, označme ako N. Druhý riadok obsahuje: nodesCount edgesCount endNodesCount cyclesCount zonesCount. Nasleduje zonesCount riadkov, kde každy riadok má nasledujúci formát: zNodesCount zEdgesCount zCyclesCount InNodesCount OutNodesCount.

To čo je ukázane vyššie je vstup pre jeden model, v prípade že je N väčšie ako 1, tak musí v rovnakej forme nasledovať popis dalsich $N - 1$ modelov.

C.3 Stavový model

Vstupný súbor

Očakávaný vstupný textový súbor má nasledujúci formát. Prvý riadok obsahuje hodnotu koľko modelov chceme vygenerovať, označme ako N. Nasleduje N riadkov, kde každý riadok obsahuje vstup pre daný model vo formáte: nodesCount edgesCount endNodesCount cyclesCount startStatesCount testStartsCount testEndsCount testStartsEndsCount. Očakávané sú prirodzené čísla podľa obmedzení z 5.2.1 oddelené medzerami. Oproti procesnému modelu je jediný rozdiel v definovaní počiatočných stavov(startStatesCount) a ďalej *testFlags* tým odpovedá: testStartsCount, testEndsCount, testStartsEndsCount.

C.4 Rozšírenie modelu

Očakávaný vstupný textový súbor má nasledujúci formát. Prvý riadok obsahuje vstup pre daný model vo formáte: nodesCount edgesCount endNodesCount cyclesCount. Očakávané sú prirodzené čísla podľa obmedzení z 5.4 oddelené medzerami.