VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

OVĚŘOVÁNÍ PARAMETRICKÝCH VLASTNOSTÍ NAD ZÁZNAMY BĚHŮ PROGRAMŮ

PARAMETRIC PROPERTIES FOR LOG CHECKER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **FILIP MUTŇANSKÝ**
AUTHOR

VEDOUCÍ PRÁCE — **ALEŠ SMRČKA, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Mutňanský Filip, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Ověřování parametrických vlastností nad záznamy běhů programů**
Parametric Properties for Log Checker
Kategorie: Analýza a testování softwaru
Zadání:

1. Seznamte se s verifikací programu za běhu. Nastudujte nástroj log-checker z platformy Testos. Seznamte se s různými možnostmi specifikace sekvence událostí založených na regulárních výrazech nebo výrazech temporálních logik.
2. Navrhněte jazyk pro specifikaci parametrických sekvencí událostí. Navrhněte rozšíření nástroje log-checker, které umožní kontrolovat splnění či porušení parametrických sekvencí událostí.
3. Implementujte rozšíření jako refaktorovaný kód nástroje log-checker.
4. Ověřte funkcionalitu nového nástroje pomocí automatizovaných testů základní funkcionality.

Literatura:

- Klaus Havelund, Giles Reger, Daniel Thoma, Eugen Zalinescu: Monitoring Events that Carry Data. Lectures on Runtime Verification 2018: 61-102.
- Domovská stránka projektu log-checker, url: https://pajda.fit.vutbr.cz/testos/log-checker
- Domovská stránka projektu Kint, url: https://kint-php.github.io/kint/
- Modul trace pro Python 2.x, url: https://docs.python.org/2/library/trace.html

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 3. června 2020
Datum schválení: 31. října 2019

Abstrakt

Cieľom tejto práce je vytvorenie nástroja, ktorý dokáže na základe užívateľom definovaných vlastností overiť, či postupnosť udalostí v záznamoch behu programu, alebo v log súbore vyhovuje definovaným vlastnostiam. Na definíciu vlastností sú použité rozšírené regulárne výrazy. Nástroj vie overovať parametrické vlastnosti. Užívateľ môže definovat závislosti medzi parametrami udalostí. Vstupom výsledného nástroja je definícia vlastností a obmedzení nad parametrami. Výstupom je report o porušených vlastnostiach s uvedenými sekvienciami udalostí, ktoré chybu spôsobili.

Abstract

The goal of this thesis is to implement a tool that based on user defined properties can verify sequences of events in the traces of the program, or the log file. Properties are defined in extended regular expressions. The tool is able to verify parametric properties. User can define relations between parameters of events. Input of this tool is the definition of properties and constraints of parameters. Output of the tool is the report of violated properties with its sequences of events that caused the error.

Kľúčové slová

verifikácia programu za behu, temporálna logika, regulárny výraz, konečný automat, Büchiho automat, Testos

Keywords

runtime verification, temporal logic, regular expression, finite automaton, Büchi automaton, Testos

Citácia

MUTŇANSKÝ, Filip. *Ověřování parametrických vlastností nad záznamy běhů programů.* Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Aleš Smrčka, Ph.D.

Ověřování parametrických vlastností nad záznamy běhů programů

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Smrčky. Uviedol som všetky literárne pramene, publikácie a dalšie zdroje, z ktorých som čerpal.

.....
Filip Mutňanský
3. júna 2020

Poďakovanie

Ďakujem pánovi Alešovi Smrčkovi za cenné rady pri tvorbe tejto práce

Obsah

Kapitola 1

Úvod

Táto práca sa venuje verifikácií programu za behu. Ide o odvetvie testovania, ktoré sa sústreďuje na správanie programu za behu. Pri verifikácií sa podľa formálnej špecifikácie vlastností rozhoduje, či je daná postupnosť udalostí v zázname programu správna, alebo nie. Cieľom tejto práce je vytvoriť nástroj *plogchecker*, ktorý dokáže overovať parametrické vlastnosti nad záznamami behu programu.

Nástroj používa na popis sekvencií udalostí regulárne výrazy. Tieto regulárne výrazy sú transformované na deterministické konečné automaty, ktoré prijímajú udalosti zo záznamu programu. Pomocou týchto konečných automatov je možné rozhodnúť, či daná postupnosť udalostí spĺňa, alebo nespĺňa odpovedajúcu vlastnosť. Výstupom nástroja *plogchecker* je report, kde sú znázornené postupnosti udalostí, ktoré porušujú definované vlastnosti.

Výsledný nástroj dokáže zachytiť hodnoty parametrov v udalostiach. Nad týmito hodnotami parametrov je možné definovať obmedzenia. Užívateľ môže definovať rovnosti parametrov medzi udalosťami. Nástroj vyhodnotí, či daná sekvencia udalostí s danými hodnotami parametrov spĺňa definované vlastnosti a obmedzenia.

Popis verifikácie programu za behu, spôsoby špecifikácie vlastností a ich výpočetné modely je v kapitole 2. Existujúce nástroje venujúce sa analýze záznamov sú priblížené v kapitole 3. Popis technológií použitých pri implementácií nástroja *plogchecker* je v kapitole 4. Špecifikácia požiadaviek pre výsledný nástroj je v kapitole 5. Návrh nástroja je v kapitole 6. Popis implementačných detailov je v kapitole 7. Testovanie nástroja je priblížené v kapitole 8.

Kapitola 2

Verifikácia programu za behu

Verifikácia programu za behu, tiež nazývaná runtime verifikácia, alebo dynamická analýza je označenie pre analýzu záznamov behu programu, alebo systému za účelom overenia správnosti. Hlavným zameraním verifikácie programu za behu je overenie, či sekvencia udalostí spĺňa, alebo porušuje špecifikované vlastnosti. Tieto vlastnosti môžu byť vyjadrené formalizmami ako konečný automat, regulárny výraz, výraz temporálnej logiky, bezkontextová gramatika, binárny rozhodovací diagram atď., alebo ich kombináciami a rozšíreniami.

Sledovaný systém môže byť softwarový program, hardware, alebo akýkoľvek systém na ktorom sa dajú sledovať dynamické vlastnosti. Na rozdiel od formálnej verifikácie, ktorá overuje celý stavový priestor systému, runtime verifikácia sleduje len jeden beh programu a pracuje priamo so sledovaným systémom. Systém je možné sledovať počas behu, alebo po dokončení (napr. analýza logov).

Podľa [9] je runtime verifikácia rozdeliteľná do 4 fáz:

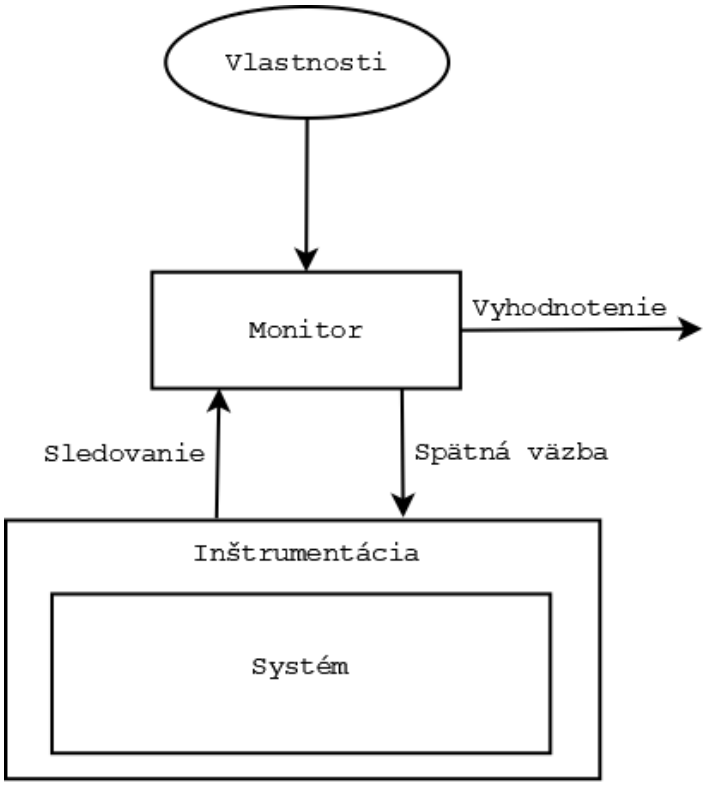
1. *Vytvorenie monitoru*: Monitor je vytvorený zo špecifikovaných vlastností.
2. *Inštrumentácia*: Napojenie monitoru na sledovaný systém, ktorý bude generovať udalosti pre monitor.
3. *Spustenie*: Systém je spustený a monitor sleduje udalosti generované systémom.
4. *Odozva*: Monitor vyhodnotí, či boli špecifikované vlastnosti splnené, alebo porušené. Monitor môže zaslať systému spätnú väzbu a na základe nej môže systém vykonať konkrétne opatrenia.

V prvej fáze sa vytvára monitor zo špecifikácie vo formálnom jazyku. Formálnym jazykom môže byť napríklad regulárny výraz, alebo výraz v temporálnej logike. V prípade, že je špecifikácia zadaná regulárnym výrazom, tak monitor odpovedá konečnému automatu. Ak sú vlastnosti zadané výrazom v temporálnej logike, tak je tento výraz transformovaný do Büchiho automatu. Podrobnejší popis formálnych jazykov vhodných pre popis vlastností je v kapitolách 2.1 a 2.2.

Inštrumentácia v druhej fáze je proces, kedy sa zo správania sledovaného systému generujú udalosti relevantné pre monitor. Prístup k inštrumentácií záleží od sledovaného systému. Mnohokrát ide len o rozhranie medzi systémom a monitorom.

Udalosti sú akékoľvek pozorovateľné zmeny v sledovanom systéme v čase (napr. zmena vnútorného stavu systému, vstup/výstup systému, atď.). Zjednodušene, udalosťou môže byť pomenovaná zmena v systéme, napríklad `fileOpen`, `fileClose`, alebo `buttonPressed`. V komplexnejšom prípade môže byť udalosť štrukturovaná, obsahovať dáta s hodnotami,

ktoré sa môžu v priebehu behu systému meniť. Tieto hodnoty je potrebné vedieť získavať a ďalej s nimi pracovať. O parametrických vlastnostiach udalostí je viac v kapitole 2.6.



Obr. 2.1: Schéma runtime verifikácie podľa [9]

Udalosti sú abstrakcia jednej pozorovateľnej zmeny v systéme. Abstrakciou jedného behu celého systému je záznam. Záznam je konečná postupnosť udalostí. Existuje viac spôsobov ako štrukturovať záznam behu programu. Prvý spôsob predpokladá, že každý prvok záznamu obsahuje len jednu udalosť. Druhý spôsob povoľuje súbežný výskyt viacerých udalostí. Voľba štrukturovania záznamu má dopad na formálnu špecifikáciu vlastností. Pre jednoduchosť budeme ďalej v texte používať prvý spôsob štrukturovania, kde poradie udalostí značí ich poradie výskytu v čase.

Poradie udalostí vyjadruje následnosť v čase. Nevyjadruje však množstvo času ubehnutého medzi udalosťami. Existujú metódy, ktoré umožňujú definovať čas v špecifikácií vlastností a tým vyjadriť napríklad dĺžku trvania udalostí.

Vlastnosť systému je v podstate množina záznamov. Špecifikácia vlastnosti je textový popis vlastnosti systému z ktorej ide odvodiť množinu záznamov.

2.1 Temporálna logika

Jednou z možností ako špecifikovať vlastnosti je temporálna logika. Temporálna logika je rozšírenie výrokovej a predikátovej logiky o čas. Jej výrazmi dokážeme vyjadriť výroky vyjadrujúce časové závislosti (napr. "Od teraz bude *stále* pršať", "Niekedy zaprší")[3]. Základné dva operátory prítomne takmer vo všetkých druhoch temporálnych logík sú:

- $\Diamond$ – *eventually* (niekedy v budúcnosti)

- $\Box$ – *always* (stále v budúcnosti)

Pohľad na čas v temporálnych logikách môže byť lineárny, alebo vetvený. V lineárnom pohľade existuje pre každý moment len jeden nasledujúci moment, vo vetvenom môže existovať viacej alternatívnych vývojov. Lineárny pohľad využíva lineárna temporálna logika (LTL). Vetvený prístup používa CTL logika (eng. Computational Tree Logic). V tejto práci je bližšie skúmaná len LTL logika.

Temporálna logika umožňuje špecifikovať len poradie udalostí, trvanie jednotlivých udalostí, alebo interval medzi udalosťami nejde v základnej temporálnej logike špecifikovať.

2.1.1 Lineárna temporálna logika

Najviac používaným druhom temporálnej logiky pri verifikácií za behu je lineárna temporálna logika (ang. Linear Temporal Logic, používa sa skratka LTL). LTL používa lineárny pohľad na čas, to znamená, že každý moment má len jeden nasledujúci moment. LTL je môžné použiť na popis synchrónneho systému, kde každá komponenta postupuje krok za krokom, kde každý krok reprezentuje postup o jednu časovú jednotku. Čas je reprezentovaný diskrétne, to znamená, že prítomný moment odpovedá súčasnému stavu a nasledujúci moment odpovedá bezprostredne nasledujúcemu stavu.

Základná LTL logika pozostáva z výrokových premenných, logických operátorov $\wedge$ (konjunkcia) a $\neg$ (negácia) a temporálnych operátorov $\bigcirc$ (*next*) a $\mathcal{U}$ (*until*). Výrokové premenné a logické operátory majú rovnaký význam ako pri bežnej výrokovej logike. Temporálny operátor $\bigcirc$ je unárny operátor vyžadujúci LTL výraz ako argument. Formula $\bigcirc\varphi$ platí, ak φ platí v nasledujúcom stave. Operátor $\mathcal{U}$ je binárny operátor, ktorý má za operandy dva LTL výrazy. Formula $\varphi_1\mathcal{U}\varphi_2$ platí, ak existuje nejaký moment v budúcnosti pre ktorý platí φ_2 a φ_1 do toho momentu platil vo všetkých predchádzajúcich momentoch [3].

Priorita unárnych operátorov je vyššia ako binárnych. Operátory $\bigcirc$ a $\mathcal{U}$ majú rovnakú prioritu. Temporálny binárny operátor $\mathcal{U}$ má vyššiu prioritu ako logické $\wedge$, $\vee$ a $\Rightarrow$.

Použitím logických operátorov $\neg$ a $\wedge$ idú odvodiť ostatné operátory výrokovej logiky. Logické operátory $\wedge$ (disjunkcia), $\Rightarrow$ (implikácia), $\Leftrightarrow$ (ekvivalencia) a $\oplus$ (exkluzívna disjunkcia) sú odvodené nasledovne:

$$\begin{aligned}
\varphi_1 \vee \varphi_2 &\equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\
\varphi_1 \Rightarrow \varphi_2 &\equiv \neg\varphi_1 \vee \varphi_2 \\
\varphi_1 \Leftrightarrow \varphi_2 &\equiv (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1) \\
\varphi_1 \oplus \varphi_2 &\equiv (\varphi_1 \wedge \neg\varphi_2) \vee (\varphi_2 \wedge \neg\varphi_1)
\end{aligned}$$

Použitím operátora *until* je možné odvodiť temporálne vlastnosti $\Diamond$ (*eventually*) a $\Box$ (*always*) nasledovne:

$$\begin{aligned}
\Diamond\varphi &\equiv true \cup \varphi \\
\Box\varphi &\equiv \neg\Diamond\neg\varphi
\end{aligned}$$

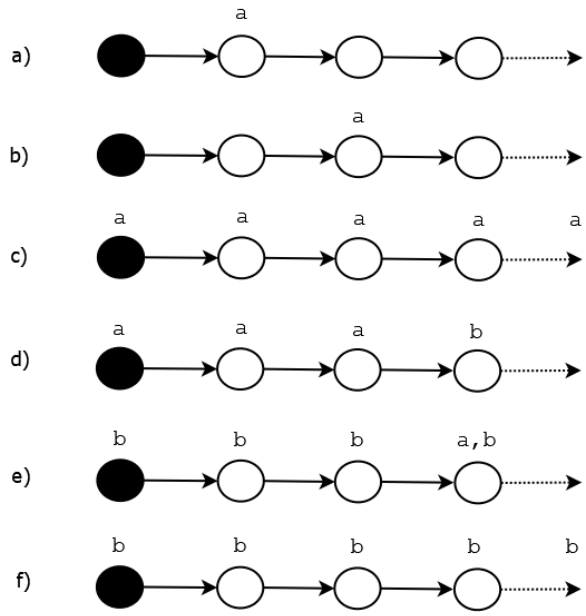
Výraz $\Diamond\varphi$ platí v prípade, že φ bude platiť niekedy v budúcnosti. Formula $\Box\varphi$ je platná v prípade, že nenastane situácia kedy bude platiť $\neg\varphi$. Inými slovami, táto formula platí, pokiaľ φ platí od prítomného momentu vždy.

Pomocou operátora *until* je možné odvodiť novú binárnu temporálnu operáciu R (*release*) nasledovne:

$$\varphi_1 R \varphi_2 \equiv \neg(\neg\varphi_1 \cup \neg\varphi_2)$$

Aby bola formula $\varphi_1 R \varphi_2$ platná, tak musí byť φ_2 platná dokým nezačne platiť φ_1. Ak nezačne platiť φ_1 nikdy, tak musí φ_2 platiť donekonečna.

Je možné definovať mnoho temporálnych operátorov (vrátane vlastností definujúcich minulosť), ale pre potreby verifikácie programu za behu stačia doteraz definované operátory. Sémantika definovaných temporálnych operátorov v tejto práci je graficky znázornená na obrázku 2.2.



Obr. 2.2: Grafické znázornenie temporálnych operácií. Každý kruh predstavuje jeden moment v čase. Čierne kruhy predstavujú prítomný moment. Prerušovaná šípka na konci znamená nekonečno. **a)** $\bigcirc a$ (*next*) **b)** $\Diamond a$ (*eventually*) **c)** $\Box a$ (*always*) **d)** $a \mathcal{U} b$ (*until*) **e)** $a \mathcal{R} b$ (*release*) situácia kedy udalosť a nastane **f)** $a \mathcal{R} b$ (*release*) situácia kedy udalosť a nikdy nenastane a udalosť b musí byť platná donekonečna.

Za použitia predchádzajúcich definícií ide definovať vlastnosti založené na poradí udalostí. Príklad systému vhodného na demonštráciu popisu chronologických vlastností je semafór. Jednotlivé stavy semafóru budú nazvané *green*, *yellow*, *red*. Napríklad vlastnosť *"Ak je na semafóre červená, nasledujúca farba nemôže byť zelená"* môže byť definovaná LTL formulou nasledovne:

$$\Box(red \rightarrow \neg \bigcirc green)$$

2.1.2 Büchiho automat

Pre verifikáciu programu so špecifikovanými vlastnosťami pomocou LTL formulí, je najprv potrebné transformovať tieto formule na odpovedajúci model schopný rozhodnúť, či bola

daná formula splnená, alebo nie. Odpovedajúci model pre LTL formule je Büchiho automat. Tento typ automatu je rozšírený konečný automat o schopnosť prijímať nekonečný vstup. Tak ako pri konečnom automate, tak aj pri Büchiho existuje deterministická a nedeterministická varianta. Podľa [15] je deterministický Büchiho automat pätica $A_d = (Q, \Sigma, \delta, q_0, F)$, kde:

- Q je konečná množina stavov automatu A_d
- Σ je konečná abeceda automatu A_d
- $\delta : Q \times \Sigma \to Q$ je prechodová funkcia
- q_0 je začiatočný stav automatu A_d, $q_0 \in Q$
- $F \subseteq Q$ je množina koncových stavov

Automat A_d prijíma tie reťazce, pri ktorých prijímaní je aspoň jeden koncový stav navštívený nekonečne veľakrát.

Keď sa snažíme previesť LTL formule na Büchiho automat je vhodné použiť všeobecný Büchiho automat, ktorý je varianta nedeterministického Büchiho automatu s rozšírenou prijímajúcou podmienkou. Podĺa [14] je všeobecný Büchiho automat pätica $A_g = (Q, \Sigma, \Delta, Q_0, \mathcal{F})$, kde:

- Q je konečná množina stavov automatu A_g
- Σ je konečná abeceda automatu A_g
- $\Delta : Q \times \Sigma \to 2^Q$ je prechodová relácia
- $Q_0 \subseteq Q$ je množina začiatočných stavov automatu A_g
- $\mathcal{F}$ je prijímajúca podmienka, ktorá je tvorená prijímajúcimi množinami. Každá prijímajúca množina $F_i \in \mathcal{F}$ je podmnožinou Q

A_d prijíma tie reťazce, pri ktorých prijímaní bol navštívený nekonečne veľakrát minimálne jeden stav z každej prijímajúcej množiny.

2.1.3 Transformácia LTL formulí na všeobecný Büchiho automat

Existuje viacero algoritmov pre transformáciu LTL výrazov na všeobecný Büchiho automat. Všetky algoritmy očakávajú výraz vo forme, kde:

- všetky negácie sa nachádzajú len pred atomickým výrokom,
- vyskytujú sa len logické operátory *true*, *false*, $\wedge$ a $\vee$ a
- vyskytujú sa len temporálne operátory $\bigcirc$ (*next*), $\mathcal{U}$ (*until*) a R (*release*)

Každý výraz LTL logiky sa dá preložiť na tento tvar. Nasledujúci algoritmus je prebratý z [14]. Pre popis algoritmu je potrebné najprv definovať uzáver formule. Nech je φ LTL formula a $\neg\varphi$ negácia formule φ, potom $CL(\varphi)$ je najmenšia množina taká, že:

- $\varphi \in CL(\varphi)$

- if $\varphi_1 \in CL(\varphi)$ then $\neg\varphi_1 \in CL(\varphi)$
- if $\bigcirc\varphi_1 \in CL(\varphi)$ then $\varphi_1 \in CL(\varphi)$
- if $\varphi_1 \wedge \varphi_2 \in CL(\varphi)$ then $\varphi_1, \varphi_2 \in CL(\varphi)$
- if $\varphi_1 \vee \varphi_2 \in CL(\varphi)$ then $\varphi_1, \varphi_2 \in CL(\varphi)$
- if $\varphi_1 \mathcal{U} \varphi_2 \in CL(\varphi)$ then $\varphi_1, \varphi_2 \in CL(\varphi)$
- if $\varphi_1 \mathcal{R} \varphi_2 \in CL(\varphi)$ then $\varphi_1, \varphi_2 \in CL(\varphi)$

Automat A_φ korešpondujúci k formule φ bude mať za stavy podmnožiny $CL(\varphi)$, ktoré sú konzistentné. Tieto podmnožiny sa nazývajú atómy. Nech φ je formula, potom $CS(\varphi) \subset CL(\varphi)$ sa nazýva množina atómov formule φ a spĺňa:

- $\forall\alpha \in CL(\varphi), \alpha \in CS(\varphi) \iff \neg\alpha \notin CS(\varphi)$
- $\forall\alpha \vee \beta \in CL(\varphi), \alpha \vee \beta \in CS(\varphi) \iff \alpha \in CS(\varphi) \vee \beta \in CS(\varphi)$
- $\forall\alpha \wedge \beta \in CL(\varphi), \alpha \wedge \beta \in CS(\varphi) \iff \alpha \in CS(\varphi) \wedge \beta \in CS(\varphi)$

Korešpondujúci všeobecný Büchiho automat k formule φ je definovaný ako $A_\varphi = (CL(\varphi) \cup \{init\}, 2^{Voc(\varphi)}, \Delta, \{init\}, F)$, kde $init$ je novovytvorený začiatočný stav automatu A_φ. $Voc(\varphi)$ je "slovná zásoba" výrazu φ, ide o množinu atomických formulí výrazu φ. Prechodová relácia $\Delta = \Delta_1 \cup \Delta_2$ je definovaná ako zjednotenie Δ_1 a Δ_2, kde Δ_1 je definovaná ako:

$$\Delta_1 = \{(init, a, \beta) | \beta \cap Voc(\varphi) \subseteq a \subseteq \{p \in Voc(\varphi) | \neg p \notin \beta\}\}$$

a Δ_2 je definovaná ako:

$$\begin{aligned}(\alpha, a, \beta) \in \Delta_2 \iff ((\beta \cap Voc(\varphi) \subseteq a \subseteq \{p \in Voc(\varphi) | \neg p \notin \beta\}) \wedge \\ (\bigcirc\varphi_1 \in \alpha \iff \varphi_1 \in \beta) \wedge \\ (\varphi_1 \mathcal{U} \varphi_2 \in \alpha \iff \varphi_2 \in \alpha \vee (\varphi_1 \in \alpha \wedge \varphi_1 \mathcal{U} \varphi_2 \in \beta)) \wedge \\ (\varphi_1 \mathcal{R} \varphi_1 \in \alpha \iff \varphi_1 \wedge \varphi_2 \in \alpha \vee (\varphi_2 \in \alpha \wedge \varphi_1 \mathcal{R} \varphi_2 \in \beta))\end{aligned}$$

Prijímajúca podmienka F je definovaná ako:

$$\forall\varphi_1 \mathcal{U} \varphi_2 \in CS(\varphi), \{\alpha \in CS(\varphi) | \varphi_2 \in \alpha \vee \neg(\varphi_1 \mathcal{U} \varphi_2) \in \alpha\}$$

2.2 Regulárne výrazy

Ďalšou možnosťou ako špecifikovať vlastnosti sú regulárne výrazy. Regulárny výraz pozostáva zo sekvencie symbolov a operátorov. Nech Σ je konečná abeceda. Regulárne výrazy nad Σ sú podľa [16]:

- $\emptyset$ označuje prázdnu množinu
- ε označuje prázdny reťazec

- a pre nejaké $a \in \Sigma$, označuje jeden znak a
- ak sú r_1 a r_2 regulárne výrazy, tak $(r_1 + r_2)$ je regulárny výraz
- ak sú r_1 a r_2 regulárne výrazy, tak $(r_1.r_2)$ je regulárny výraz
- ak je r regulárny výraz, tak (r^*) je regulárny výraz

Žiadne iné regulárne výrazy mimo vymenovaných sa nedajú vytvoriť. Operátor + sa nazýva alternatíva. Výraz $r_1 + r_2$ predstavuje zjednotenie množín popísaných výrazmi r_1 a r_2. Operátor . predstavuje operáciu konkatenácie. Výraz $r_1.r_2$ označuje množinu reťazcov vytvorených konkatenovaním reťazcov z množiny r_1 s reťazcami z množiny r_2. Operátor * sa nazýva iterácia. Výraz r^* označuje množinu všetkých reťazcov (vrátane prázdneho reťazca) vytvorených konkatenáciou akéhokoľvek konečného počtu reťazcov z množiny určenou výrazom r.

Regulárny výraz rr^* sa dá zapísať ako r^+. Tento nový operátor sa nazýva pozitívna iterácia. Výraz $r_1.r_2$ sa môže zjednodušiť na r_1 Priorita operátorov používaných v regulárnych výrazoch je nasledovná. Iterácia a pozitívna iterácia majú navyššiu prioritu, nasleduje konkatenácia a najnižšiu prioritu má alternatíva.

Po definícií regulárnych výrazov ide špecifikovať chronologické vlastnosti pre verifikáciu programu za behu. Opäť na demoštráciu použijeme semafór z kapitoly 2.1.1. Vlastnosť *"Ak je na semafóre červená, nasledujúca farba nemôže byť zelená"* môžeme zapísať regulárnym výrazom nasledovne:

$$(red \mid green \mid yellow)^* \; red \; yellow$$

Každá postupnosť udalostí vyhovujúca tomuto regulárnemu výrazu bude spĺňať vlastnosť *"Ak je na semafóre červená, nasledujúca farba nemôže byť zelená"*. Je možné špecifikovať spomínanú vlastnosť aj ako výraz, ktorý keď bude splnený, tak daná postupnosť udalostí bude vyhodnotená ako nevyhovujúca.

$$(red \mid green \mid yellow)^* \; red \; green$$

Tento výraz vyjadruje situáciu kedy je semafóre červená a bezprostredne za ňou sa vyskytne zelená. Takáto postupnosť udalostí porušuje špecifikovanú vlastnosť a bude vyhodnotená ako nevyhovujúca.

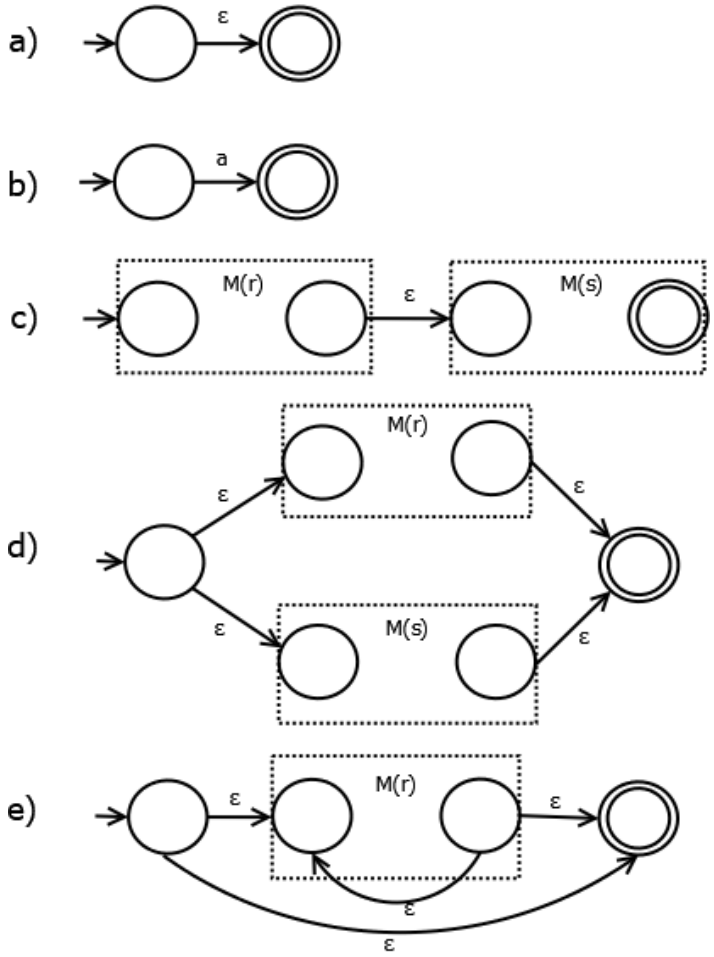
2.3 Transformácia regulárnych výrazov na konečný automat

Vlastnosť zadaná regulárnym výrazom je len textové vyjadrenie danej vlastnosti. Pre monitorovanie postupnosti udalostí je potrebné transformovať tento regulárny výraz na konečný automat. Po prevedení výrazu na konečný automat, bude automat vedieť rozhodnúť, či postupnosť udalostí vyhovuje danej vlastnosti, alebo ju porušuje. Konečný automat je podľa [16] pätica $M = (Q, \Sigma, \delta, q_0, F)$ kde:

- Q je konečná množina stavov automatu M
- Σ je konečná abeceda automatu M
- $\delta : Q \times \Sigma \to 2^Q$ je prechodová funkcia

- q_0 je začiatočný stav automatu M, $q_0 \in Q$
- $F \subseteq Q$ je množina koncových stavov

Na transformáciu sa používa rekurzívny algoritmus, ktorý najprv rozdelí výraz na atomické podvýrazy. Tieto podvýrazy sa následne prevedú podľa pravidiel na obrázku 2.3. Výsledný konečný automat je nedeterministický s ε-prechodmi.



Obr. 2.3: Znázornenie transformácie regulárneho výrazu na konečný automat. Každý konečný automat na obrázku je výsledkom transformácie z regulárneho výrazu **a)** ε **b)** a **c)** $r.s$ **d)** $r + s$ **e)** r^*. Stav automatu s vchádzajúcou šípkou je začiatočný stav výsledného konečného automatu. Stav s dvojitým obrysom je konečný stav konečného automatu. Prerušované obrysy označujú konečný automat regulárneho podvýrazu. Podvýraz je znázornený bez vnútornej štruktúry

2.4 Determinizácia konečného automatu

Nech $M = (Q, \Sigma, \delta, q_0, F)$ je nedeterministický konečný automat s ε-prechodmi. Tento typ konečného automatu však dovoľuje situáciu kedy $\exists q \in Q \ \exists a \in \Sigma : |\delta(q, a)| > 1$. Povoľuje teda prítomnosť viacerých prechodov z jedného stavu pre jeden vstupný symbol. V praktickom použití je tento typ konečného automatu nevhodný. Preto je potrebné ho transformovať

na deterministický, neobsahujúci ε-prechody. Nasledujúci spôsob prevodu je prevzatý z [16]. Nech $M' = (Q', \Sigma, \delta', q'_0, F')$ je deterministický konečný automat ekvivalentný automatu M. Algoritmus:

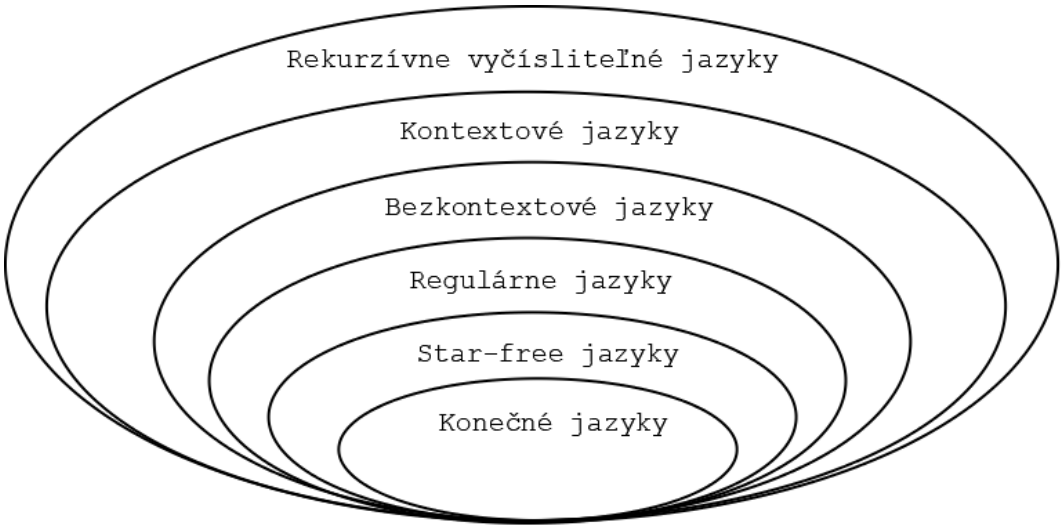
1. $Q' = 2^Q \setminus \{\emptyset\}$
2. $q'_0 = \{q_0\}$
3. $F' = \{S | S \in 2^Q \wedge S \cap F \neq \emptyset\}$
4. $\forall S \in 2^Q \setminus \{\emptyset\}\ \forall a \in \Sigma$:
   - $\delta'(S, a) = \bigcup_{q \in S}\ \delta(q, a)$, ak $\bigcup_{q \in S} \delta(q, a) \neq \emptyset$
   - inak $\delta'(S, a)$ nie je definovaná
5. $F \subseteq Q$ je množina koncových stavov

Výsledný konečný automat M' spĺňa podmienku: $\forall q \in Q'\ \forall a \in \Sigma : |\delta'(q, a)| <= 1$. Teda neexistuje stav, ktorý by mal pre jeden vstupný symbol viac prechodov.

2.5 Zhrnutie spôsobov špecifikácie vlastností

Zo spôsobov špecifikácie vlastností boli v tejto práci popísané výrazy lineárnej temporálnej logiky a regulárne výrazy. Podľa [5] patria jazyky definované lineárnou temporálnou logikou medzi star-free jazyky. Star-free jazyky sú jazyky definovatelné regulárnym výrazom pozostávajucim len zo symbolov, operátorov konkatenácie a boolean operátorov (vrátane komplementu), ale neobsahujúcim operátor * (iterácia)[8]. Star-free jazyky obsahujú všetky konečné jazyky. Existujú jazyky, ktoré sú regulárne, ale nie star-free (napríklad jazyk definovaný výrazom $(aa)^*$). Na obrázku 2.4 je znázornený vzťah star-free jazykov k Chomského hierarchií jazykov.

Jazyky definované regulárnymi výrazmi patria medzi regulárne jazyky. Avšak len za použitia regulárnych výrazov z formálnej teórie jazykov bez akýchkoľvek rozšírení. Použitím regulárnych výrazov s rozšíreniami (napr. spätné odkazy) je môžné popísať jazyky silnejšie ako regulárne.



Obr. 2.4: Vzťah star-free jazykov k Chomského hierarchii

Doteraz spomenuté jazyky vhodné na špecifikáciu vlastností boli maximálne regulárne v Chomského hrierarchii. Existujú formalizmy, ktoré sú silnejšie, ale väčšinou ich vyššia výpočetná sila ide na úkor ich použiteľnosti. Napríklad bezkontextové gramatiky majú síce vyššiu výpočetnú silu, ale nezachycujú chronologické vlastnosti tak názorne ako napríklad výrazy temporálnej logiky. Jedným zo spôsobov ako zvýšiť výpočetnú silu a zachovať užívateľskú prístupnosť je napríklad kombinácia LTL formulí s regulárnymi výrazmi. Touto kombináciou je možné zvýšiť vyjadrovaciu schopnosť temporálnych formulí na regulárne jazyky a zároveň zachovať schopnosť temporálnych formulí zachytiť chronologické vlastnosti.

2.6 Parametrické vlastnosti záznamov behu programu

Doteraz boli v tejto práci udalosti brané ako atomické jednotky bez vnútornej štruktúry. Napríklad udalosti `buttonPressed`, `buttonReleased`, alebo `mouseClicked` sú atomické. Jediné čo sa dá z takýchto záznamov zistiť je aká udalosť práve nastala a podľa nej vykonať odpovedajúci krok vo výpočetnom modely.

Predmetom tejto práce sú parametrické vlastnosti udalostí, čiže na udalosti sa nahliada ako na štrukturované a obsahujúce dáta. Napríklad na udalostiach `fileOpen(file1)`, `fileOpen(file2)`, alebo `fileClose(file1)` je už možné sledovať parametrické vlastnosti. U dvojici udalostí `fileOpen(file1)` a `fileOpen(file2)` je možné vidieť, že sú to síce rovnaké udalosti, ale ich dáta sa líšia. U dvojici udalostí `fileOpen(file1)` a `fileClose(file1)` je naopak vidieť, že ide o rôzne udalosti, ale ich hodnota parametrov je rovnaká. Pri monitorovaní udalostí s parametrickými vlastnosťami je potrebné sledovať o aké udalosti ide a zároveň sledovať hodnoty dát, ktoré dané udalosti obsahujú. Majme nasledujúci záznam behu programu v chronologickom poradí:

```
fileOpen(file1)
fileOpen(file2)
fileClose(file2)
fileClose(file1)
```

Záznam `fileOpen(file1)` značí otvorenie súboru s názvom `file1`. Záznam `fileClose(file1)` značí zatvorenie súboru s názvom `file1`. Vlastnosť systému "*Každý otvorený súbor musí byť uzatvorený*" je zjavne v tomto zázname behu programu splnená. Rozpoznanie, či je táto vlastnosť splnená, je možné len parametrickým prístupom k udalostiam.

Predcházajúca vlastnosť je popísateľná minimálne bezkontextovou gramatikou, ide o typický prípad bezkontextového jazyka, ktorý nie je regulárny $L = \{a^n b^n \mid n >= 0\}$. Jednou zo základných výziev parametrického pohľadu na udalosti je špecifikácia vlastností v parametrickom prostredí.

Kapitola 3

Popis existujúcich nástrojov pre verifikáciu programu za behu

Existuje pomerne veľa nástrojov pre verifikáciu programu za behu. Väčšinou sa líšia v spôsobe špecifikovania vlastností a vo forme vstupu. Niektoré nástroje uprednostňujú vyjadrovaciu schopnosť formálnej špecifikácie na úkor užívateľskej prívetivosti, iné uprednostňujú intuitívnosť definície vlastností. V súčasnosti neexistuje jeden spôsob špecifikácie, ktorý by sa javil ako jediný správny. Nasleduje popis pár nástrojov venujúcich sa verifikácií programu za behu.

3.1 RULER

Nástroj RULER [4] používa na špecifikáciu vlastností relatívne nízkoúrovňový systém pravidiel. Nízkoúrovňová špecifikácia vlastností znamená, že už samotná špecifikácia predstavuje výpočetný model, ktorý bude rozhodovať o postupnostiach udalostí. Pravidlo v nástroji RULER môže vyzerať nasledovne:

$$Start : openFile(f : obj) \rightarrow Track(f)$$

Start je názov pravidla. Ak je toto pravidlo aktívne, tak v prípade, že nastane udalosť *openFile* s parametrom *f* tak aktivuje pravidlo *Track* s parametrom *f*. Ak nenastane udalosť *openFile*, tak sa pravidlo *Start* deaktivuje.

Definované pravidlá sú neperzistentné, to znamená, že pravidlo je aktivované pre nasledujúci krok, v ňom je použité a následne je automaticky deaktivované. Preto je jazyk nástroja RULER vhodný na interpretáciu gramatík, ale pri zadávaní pravidiel užívateľom je špecifikovanie pravidiel pomerne neintuitívne.

3.2 LOGSCOPE

LOGSCOPE [4] je monitorovací systém pôvodne vytvorený pre testovanie vozidla od NASA pre misiu Mars Rover v roku 2011. Je implementovaný v jazyku Python. Podporuje analýzu logov pre testovacie účely. Vstupom nástroja LOGSCOPE je log a špecifikácia vlastností. Výstupom je výsledný report o porušeniach vlastností. Nástroj očakáva log ako sekvenciu Python slovníkov, kde slovník predstavuje jednu udalosť.

Jazyk špecifikácie udalostí pozostáva z dvoch častí: vysoko úrovňového jazyka podobného temporálnej logike a nízko úrovňového jazyka inšpirovaného nástrojom RULER. Prvý

jazyk je prekladaný na konečné automaty. Očakáva sa, že užívateĺ bude písať špecifikáciu vlastností primárne v tomto jazyku. Druhý jazyk je používaný v prípade, že je potrebná vyššia vyjadrovacia schopnosť. Táto vyššia vyjadrovacia schopnosť je ale za cenu nižšej užívateľskej prívetivosti.

3.3 LogFire

LogFire [10] je jazyk založený na jazyku Scala, určený pre analýzu záznamov behu programu. LogFire umožňuje písať špecifikáciu vlastností vo forme pravidiel v jazyku Scala. Scala je regulérny programovací jazyk, takže v pravidlách je možné využívať akékoľvek konštruckie programovacieho jazyka. Keďže je jazyk špecifikácie vyjadriteľný programovacím jazykom, tak má rekúrzívne vyčísliteľnú vyjadrovaciu schopnosť. Nevýhodou jazyka LogFire, že autor špecifikácie vlastností musí ovládať jazyk Scala.

3.4 JavaMOP

JavaMOP je nástroj a jazyk z frameworku MOP (Monitoring-Oriented Programming) zameraný na Java aplikácie [9]. Vo frameworku MOP je podporované monitorovanie za behu. Zo špecifikácie vlastností sú automaticky vytvorené monitory a tie sú použité pri overovaní dynamických vlastností systému. JavaMOP podporuje niekoľké logické pluginy na špecifikáciu vlastností. Užívateľ si môže vybrať najvhodnejší plugin na popis danej vlastnosti. JavaMOP ponúka nasledovné pluginy: FSM (konečné automaty), ERE (rozšírené regulárne výrazy), CFG (bezkontextové gramatiky), LTL (lineárna temporálna logika), PTLTL (lineárna temporálna logika s operátormi pre minulosť) a SRS (systémy prepisovacích pravidiel).

Kapitola 4

Technológie využívané v nástroji plogchecker

4.1 Pôvodný nástroj logchecker

Nástroj *plogchecker* je postavený na existujúcom nástroji *logchecker*, ktorý bol súčasťou bakalárskej práce [12]. Táto práca sa venovala verifikácií programu za behu, výsledný nástroj však neberie do úvahy parametrické vlastnosti. Nástroj vyžaduje ako vstup súbor s definíciou vlastností vo formáte yaml a log súbor. Tento súbor obsahuje 2 časti: *properties* a *events*. V sekcii *properties* sú definované vlastnosti regulárnymi výrazmi. V sekcii *events* sú definované udalosti. Vo výpise 4.1 je popísana štruktúra tohto súboru rozvinutou Backusovou-Naurovou formou.

```
<propertyfile> ::= [<propertieslist>] [<badpropertieslist>] <eventslist>
<propertieslist> ::= properties : \n <propertydef> {<propertydef>}
<badpropertieslist> ::= badproperties : \n <propertydef> {<propertydef>}
<propertydef> ::= <id> : <regex> \n
<eventlist> ::= events : \n <eventdef> \n {<eventdef>}
<eventdef> ::= <id> : <regex> \n
<id> ::= <letter> {<idchar>}
<idchar> ::= <letter> | <digit>
```

Výpis 4.1: Štruktúra definície vlastností v rozvinutej Backusovej-Naurovej forme

Pravidlo `<regex>` značí bežný regulárny výraz. V tomto výraze sa vyskytujú identifikátory zo sekcie `events`. V súbore musí byť zadaná minimálne jedna zo sekcií `properties`, alebo `badproperties`. Tiež musí byť definovaná sekcia `events`. Na výpise 4.2 je príklad súboru s definíciou vlastností z nástroja *logchecker*.

```
properties:
  p1: "(de)*"

badproperties:
  p2: "a|b|c"

events:
```

```
a: "^.*OPEN UDP 172\\.20\\.73\\.241 239\\.255\\.255\\.250.*$"
b: "^.*OPEN UDP 192\\.168\\.99\\.165 239\\.255\\.255\\.250.*$"
c: "^.*OPEN UDP 192\\.168\\.183\\.114 239\\.255\\.255\\.250.*$"
d: "^.*OPEN TCP 192.168.72.12 10.20.158.58 3687 80.*$"
e: "^.*CLOSE TCP 192.168.72.12 10.20.158.58 3687 80.*$"
```

Výpis 4.2: Príklad súboru s definíciou vlastností prebratý z [12]

Vlastnosť `p1` značí: *"TCP spojenie medzi 192.168.72.12 a 10.20.158.58 musí vždy byť ukončené"*, táto vlastnosť je v sekcii `properties` to znamená, že musí byť splnená, aby nenastala nežiadúca postupnosť. Vlastnosť `p2` značí: *"Nemôže byť prijatý žiadny packet z menovaných IP adries"*, je súčasťou sekcie `badproperties`, teda ak postupnosť udalostí vyhovuje tomuto výrazu, tak nastala chyba.

Z definície vlastností sú vytvorené konečné automaty. Každá vlastnosť je reprezentovaná jedným konečným automatom. Následne je spustený monitor, ktorý sleduje udalosti (záznamy v log súbore, alebo na presmerovanom vstupe) a po jednej ich spracováva. Keď nastane udalosť, tak sa všetky automaty pokúsia vykonať prechod podľa definície. Keď sa monitor dostane na koniec log súboru, tak sa vyhodnotia všetky vlastnosti. Aby boli vlastnosti zo sekcie `properties` splnené, musia byť na konci monitorovania prislúchajúce konečné automaty v koncovom stave. Pre vlastnosti zo sekcie `badproperties` platí opak. Beh programu je úspešný, pokiaľ žiadny automat z `badproperties` neskončil v koncovom stave.

Výstupom tohto nástroja je report vo formáte json o porušených sekvenciach udalostí. Na 4.3 je príklad výstupného reportu nástroja *logchecker*.

```
{
   "badproperties": {
      "p2": {
         "property": "a|b|c",
         "violated": [
            {
               "id": "automaton_6",
               "is_property_met": true,
               "events_sequence": [
                  {
                     "event_id": "c",
                     "log_file": "tests/logs/win_firewall",
                     "log_lineno": 84,
                     "log_line": "OPEN UDP 192.168.183.114 239.255.255.250 ..."
                  }
               ]
            }
         ]
      }
   }
}
```

Výpis 4.3: Príklad výstupného reportu nástroja *logchecker* s porušenou vlastnosťou `p2`. Prebratý z [12]

4.1.1 Nedostatky nástroja logchecker

Ako už bolo spomenuté, *logchecker* sa nezaoberá parametrickými vlastnosťami udalostí. Všetky udalosti, ktoré v sledovanom systéme nastanú berie ako atomické. Udalosti sa nedajú definovať všeobecne, musia byť zadané presne bez použitia parametrov. Napríklad v 4.2 nejde udalosti zovšeobecniť pre všetky IP adresy.

Pri veľkých množstvách záznamov programu a definovaných udalostí začne byť verifikácia za behu náročná na pamäť. Preto by bolo vhodné vytvoriť mechanizmus podobný garbage collection ktorý bude sledovať aktuálne automaty a uvoľňovať tie dlho bežiace. Nástroj by mal dať vedieť užívateľovi, že takúto akciu vykonal.

4.2 YACC

YACC je nástroj pôvodne určený pre Unix, ktorý z užívateľom zadanej bezkontextovej gramatiky dokáže vytvoriť vlastný parser [11]. *Plogchecker* používa PLY (Python-Lex-Yacc), čo je varianta YACC implementovaná v jazyku Python. PLY využíva LR (Left-To-Right) parsovanie narozdiel od YACC, ktorý využíva LALR (Look-Ahead LR) [6]. V nástroji *plogchecker* je PLY používaný na vytvorenie parseru vlastností, udalostí a obmedzení.

4.3 Automata

Automata je Python knižnica ktorá implementuje algoritmy a štruktúry pre konečný automat [7]. Nástroj *plogchecker* využíva túto knižnicu pre vytvorenie nedeteministického konečného automatu s ε-prechodnmi a následne na prevod tohto automatu na deterministický konečný automat.

Kapitola 5

Špecifikácia požiadaviek nástroja plogchecker

Tabuľka 5.1: Špecifikácia požiadaviek

Identifikátor	Popis	Závislosť
01	Užívateľ môže definovať vlastnosti nad sekvenciami záznamov v danom log súbore	
02	Možnosť definovať parametrické vlastnosti nad udalosťami	01
03	Možnosť definovať obmedzenia nad parametrami udalostí - rovnosť parametrov medzi dvoma udalosťami	02
04	Možnosť definovania komplikovanejších obmedzení (napr. $a.1 < 5$)	03
05	Výstupom bude report, kde bude informácia o sekvencii udalostí, ktorá spôsobila chybu	01
06	Priebežné hlásenie porušených vlastností (len negatívne vlastnosti)	05
07	Nástroj musí podporovať akýkoľvek formát záznamov	01
08	Záznamy môžu byť zadávané aj na štandartný vstup	01
09	Garbage collector - priebežné uvoľňovanie pamäte počas verifikácie mazaním otvorených inštancií	01
10	Udržiavanie informácií o dobe po ktorú bola inštancia otvorená	09
11	Pred zmazaním otvorenej inštancie vypísať varovanie o potenciálnom zmazaní	10

Kapitola 6

Návrh nástroja plogchecker

Táto kapitola sa venuje návrhu nástroja plogchecker. Všeobecný náhľad na výsledný systém a tok dát medzi jednotlivými komponentami je v kapitole 6.1. Štruktúra súboru s definíciou vlastností je zobrazená v kapitole 6.2. Spracovanie jednotlivých sekcií súboru je priblížené v podkapitolách 6.2.1 (*properties* a *badproperties*), 6.2.3 (*events*) a 6.2.4 (*constraints*). Generovanie konečných automatov je znázornené v 6.2.2. Obsah finálneho reportu o porušených vlastnostiach vo formáte JSON je v kapitolle 6.4. Priebežné uvoľňovanie prostriedkov pre oddialenie spadnutia nástroja je popísané v kapitole 6.6. Analýza časovej a priestorovej zložitosti je v kapitolách 6.7 a 6.8.

6.1 Návrh toku dát

Na obrázku 6.1 je znázornený tok dát medzi jednotlivými komponentami v nástroji *plogchecker*. Vstupmi nástroja sú *definícia vlastností* a *log súbor*. *Definícia vlastností* je súbor kde sú špecifikované vlastnosti (pozitívne a negatívne), udalosti a obmedzenia nad parametrami udalostí.

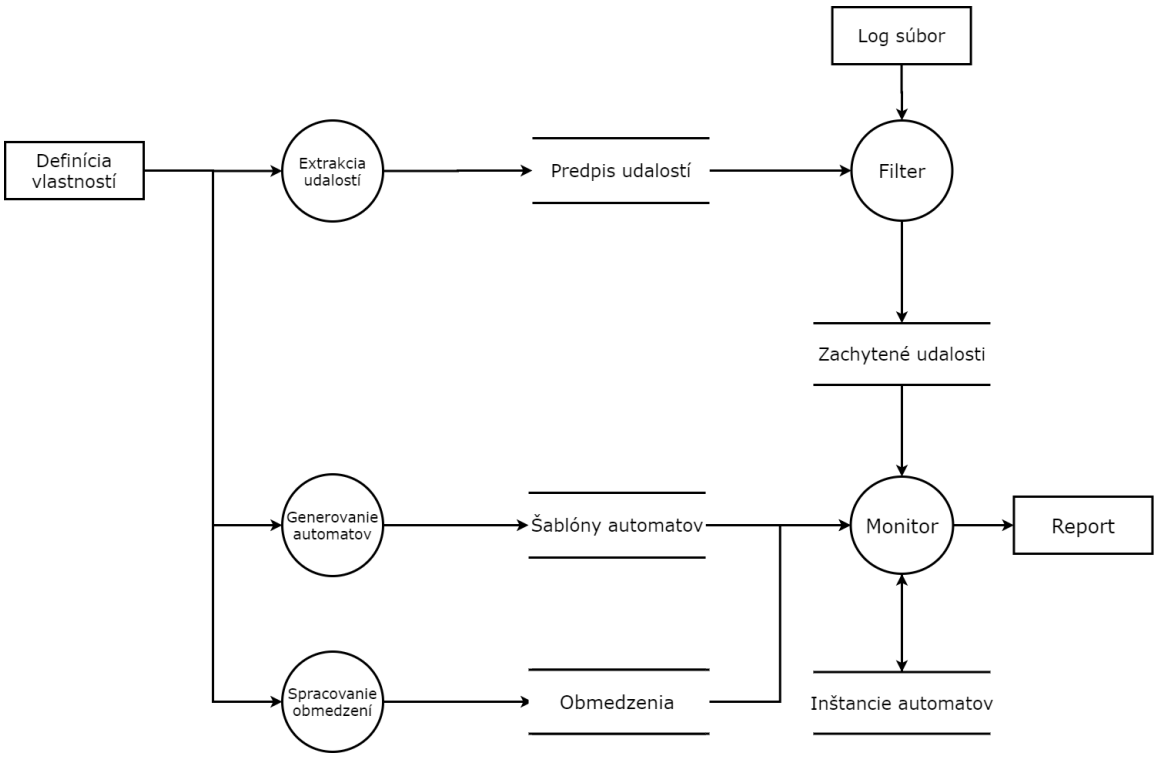
Z tejto špecifikácie sú z definície vlastností vytvorené šablóny konečných automatov (proces *generovanie automatov* v diagrame) a z definície udalostí vytvorený *predpis udalostí* (proces *extrakcia udalostí*). *Predpis udalostí* je zoznam regulárnych výrazov značiacich jednotlivé udalosti a ich identifikátory. Definícia obmedzení je procesom *spracovanie obmedzení* transformovaná do *obmedzení*, ktoré môže *monitor* počas behu overovať.

Filter je proces, ktorý vyberá z log súboru podľa *predpisu udalostí*, udalosti, ktoré vyhovujú danému regulárnemu výrazu a sú relevantné pre verifikáciu. Počas behu verifikácie monitor čaká na príchod udalostí od *filtru*.

Ak *monitor* zaznamená udalosť a na nejakej *šablóne konečného automatu* je možné vykonať prechod zo začiatočného stavu, tak monitor vytvorí novú inštanciu konečného automatu a vykoná na nej daný prechod. Inštancie automatov sú uchovávané v zozname *inštancie automatov*. S príchodom novej udalosti sa *monitor* tiež pokúša vykonať prechody na už existujúcich inštanciach automatov. Pri overovaní prechodov *monitor* kontroluje, či je v odpovedajúcom automate k dispozícií prechod z aktuálneho stavu a zároveň overuje, že sú splnené všetky *obmedzenia* nad parametrami. Ak sa niektorá z inštancií konečných automatov dostane do koncového stavu, zo zoznamu je vymazaná.

Keď proces *filter* dojde na koniec log súboru, *monitor* vyhodnotí podľa stavu inštancií konečných automatov, či boli definované vlastnosti splnené, alebo nesplnené. Na záver

verifikácie je vytvorený *report*, kde sú vo formáte JSON popísané postupnosti porušených vlastností.



Obr. 6.1: Data-flow diagram znázorňujúci vstupy a výstupy nástroja *plogchecker*. Použitá Yourdon/DeMarco notácia.

6.2 Definícia vlastností

Nástroj *plogchecker* má podobnú štruktúru ako *logchecker* [12]. Definícia vlastností bola rozšírená o sekciu constraints a zachováva syntax YAML súborov. Definícia vlastností popísaná Backus-Naur formou je v 6.1.

```
<propertyfile> ::= [<propertieslist>] [<badpropertieslist>]
                   <eventslist> [<constraintslist>]
<propertieslist> ::= properties : \n <propertydef> {<propertydef>}
<badpropertieslist> ::= badproperties : \n <propertydef> {<propertydef>}
<propertydef> ::= <letter> : <regex> \n
<eventlist> ::= events : \n <eventdef> \n {<eventdef>}
<eventdef> ::= <letter> : <regex> \n
<constraintslist> ::= constraints : \n <constraintdef> {<constraintdef>}
<constraintdef> ::= <equal_expr> | <comparison_expr>
<equal_expr> ::= - <position_par> = <position_par>
                     {= <position_par>}\n
<comparison_expr> ::= <position_par>|<number> <op> <position_par>|<number>\n
<position_par> ::= <letter> . <number>
```

```
<op> ::= '<' | '>' | '<=' | '>='
```

Výpis 6.1: Štruktúra definície vlastností v rozvinutej Backus-Naur forme

Súbor s vlastnosťami musí obsahovať aspoň jednu zo sekcií *properties*, alebo *badproperties*. Tu sú špecifikované postupnosti udalostí rozšírenými regulárnymi výrazmi. Sekcia *properties* obsahuje postupnosti ktoré musia nastať, sekcia *badproperties* obsahuje tie postupnosti, ktoré nastať nemôžu. Ďalej nasleduje sekcia *events*, kde sú definované udalosti regulárnymi výrazmi. Každej udalosti odpovedá jeden identifikátor a jeden regulárny výraz, ktorý je pri fáze filtrovania aplikovaný na riadky log súboru. Na konci definíčného súboru je sekcia *constraints*, kde sú definované parametrické závislosti medzi udalosťami.

```
properties:
  p1: abc

events:
  a: foo\((.+)\)
  b: bar\((.+)\)
  c: baz\((.+),(.+)\)

constraints:
  - a.1 = c.1
  - b.1 = c.2
```

Výpis 6.2: Príklad definície vlastností a obmedzení v nástroji *plogchecker*

Príklad súboru s parametrickými vlastnosťami je v 6.2. Tento príklad obsahuje len jednu pozitívnu vlastnosť *p1*. Vlastnosť *p1* značí, že monitor očakáva výskyt udalosti s identifikátorom *a* nasledovanú udalosťou *b* a udalosťou *c*.

Na zachytenie dát nesených udalosťami sú použité okrúhle zátvorky (v 6.2 zvýraznené farbou) v regulárnom výraze definujúcom udalosť (*event*). Okrúhle zátvorky v regulárnom výraze sú interpretované ako pozičné parametry. Toto umožňuje indexovať jednotlivé parametre v definícií obmedzení.

Obmedzenia nad parametrami sú definované v sekcií *constraints*. Obmedzenia sú definované pomocou pozičných parametrov. Napríklad obmedzenie $b.1 = c.2$ značí, že prvý parameter udalosti *b* má byť rovnaký ako druhý parameter udalosti *c* vo všetkých výskytoch týchto udalostí. Obmedzenia sú popísané bližšie v kapitole 6.2.4

6.2.1 Spracovanie sekcie properties a badproperties

Udalosti sú v sekcií *properties* a *badproperties* definované ako dvojica $id : regex$, kde *id* je identifikátor udalosti a *regex* je definícia postupnosti udalostí. Identifikátor môže byť malé, alebo veľké písmeno anglickej abecedy. Sekvencie udalosti v sekcií *properties* sú vlastnosti, ktoré sa musia v priebehu monitorovania vyskytnúť, v prípade, že nenastanú, nástroj zahlási nedokončenú sekvenciu vo finálnom reporte. Postupnosti udalostí definované v *badproperties* sú naopak vlastnosti, ktoré nemôžu v priebehu monitorovania nastať, ak nastanú, nástroj túto sekvenciu zahlási v reporte.

Pre špecifikovanie postupnosti udalostí je použitá syntax rozšírených regulárnych výrazov [2]. Syntax rozšírených regulárnych výrazov umožňuje definovať operátory bez spätného lomítka a obsahuje operátor alternácie. Definícia vlastností podporuje päť typov operátorov:

- konkatenácia (”)
- alternácia (’|’)
- iterácia (’*’)
- pozitívna iterácia (’+’)
- obmedzená iterácia (’{}’)

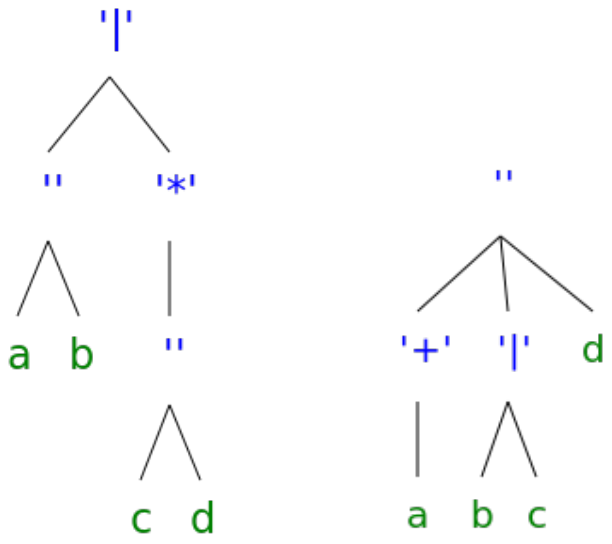
Operátor konkatenácie je reprezentovaný prázdným znakom a značí výskyt dvoch udalostí chronologicky za sebou. Operátory alternácie, iterácie a pozitívnej iterácie majú rovnaký význam a definíciu ako pri rozšírených regulárnych výrazoch.

Obmedzená iterácia označuje presný počet opakovaní v iterácií. Napríklad $a\{3\}$ značí výskyt udalosti a trikrát bezprostredne za sebou. Je podporovaná aj definícia intervalu v obmedzenej iterácií. Napríklad $a\{2,4\}$ označuje výskyt udalosti a dvakrát, trikrát, alebo štyrikrát za sebou.

```
regex = branch [ { | branch } ]
branch = { expr }
expr = ’(’ regex ’)’ | expr iter | EVENT
iter = ’*’ | ’+’ | ’{’ n ’}’ | ’{’ n1 , n2 ’}’ | ε
```

Výpis 6.3: Gramatika popisujúca rozšírené regulárne výrazy vyjadrujúce postupnosti udalostí v rozšírenej Backus-Naur forme. Prebraté z [12]

Pri spracovaní rozšírených regulárnych výrazov definujúcich vlastnosti je použitá gramatika 6.3. Z každého regulárneho výrazu je vytvorený abstraktný syntaktický strom, z ktorého je následne vygenerovaný konečný automat. Príklady odpovedajúcich abstraktných stromov pre regulárne výrazy sú na obrázku 6.2.

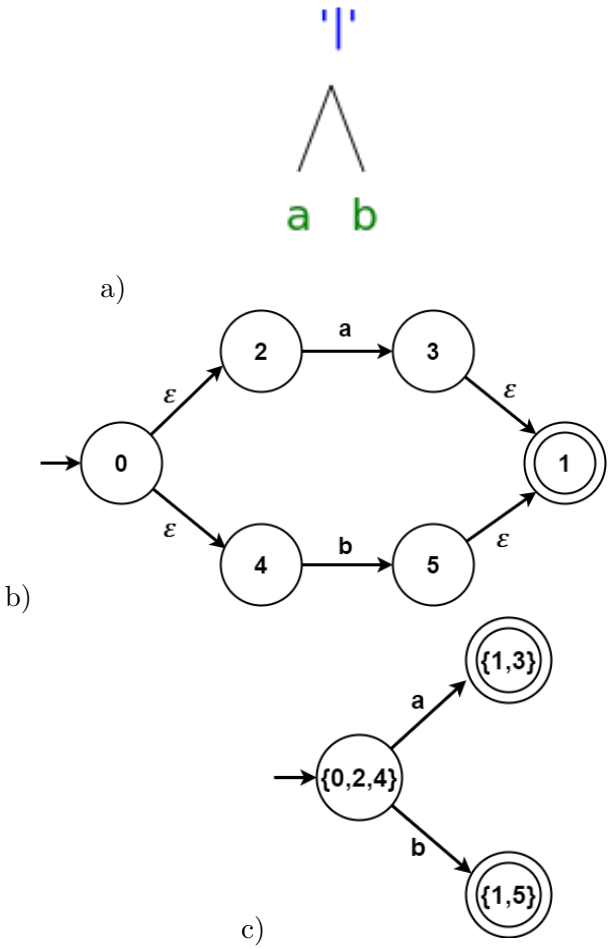


a) b)

Obr. 6.2: a) abstraktný syntaktický strom pre rozšírený regulárny výraz $(ab)|(cd)*$ b) abstraktný syntaktický strom pre rozšírený regulárny výraz $a + (b|c)d)*$

6.2.2 Generovanie konečných automatov

Z každého abstraktného syntaktického stromu je vytvorený konečný automat odpovedajúci pôvodnému regulárnemu výrazu. Pre prevod abstraktného syntaktického stromu na nedeterministický konečný automat s ε-prechodmi je použitá metóda popísaná v kapitole 2.3.



Obr. 6.3: a) abstraktný syntaktický strom pre regulárny výraz $a|b$ b) nedeterministický konečný automat s ε-prechodmi odpovedajúci regulárnemu výrazu $a|b$ c) determinický konečný automat odpovedajúci regulárnemu výrazu $a|b$

Na obrázku 6.3 je znázornený postup vytvárania deterministického konečného automatu pre jednoduchú vlastnosť definovanú regulárnym výrazom $a|b$. Najprv je vytvorený abstraktný syntaktický strom (a) odpovedajúci danému regulárnemu výrazu. Následne je syntaktický strom prevedený na nedeterministický konečný automat s ε-prechodmi (b) metódou popísanou v kapitole 2.3. Identifikátory stavov sú vpísané do jednotlivých stavov. Potom je transformovaný nedeterministický konečný automat na deterministický (c) metódou popísanou v kapitole 2.4. Finálny deterministický automat je vytvorený zhlukovaním stavov nedeterministického automatu so zachovaním množiny prijímaných reťazcov. Identifikátory deterministického konečného automatu sú na obrázku 6.3 vyjadrené ako množiny. Prvky množín sú identifikátory stavov z predošlého nedeterministického automatu.

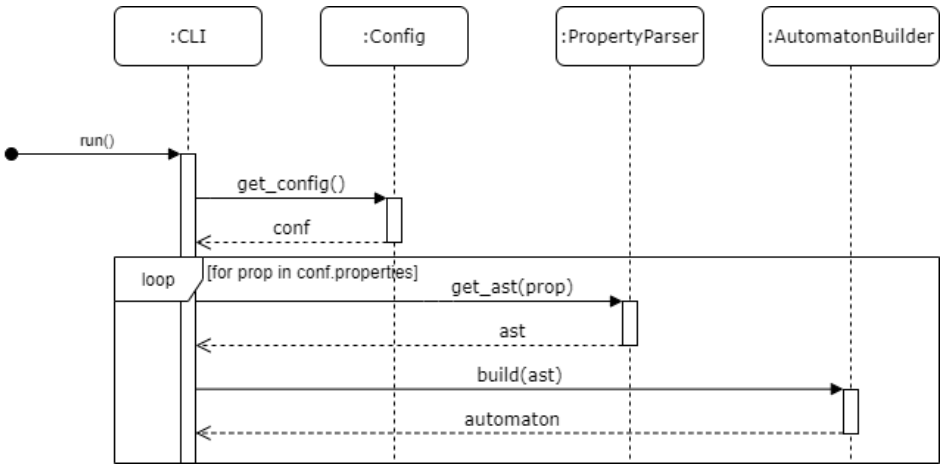
Vlastnosti môžu byť zadané aj komplexnými regulárnymi výrazmi obsahujúcimi niekoľko podvýrazov (napríklad 6.2). V tom prípade je pri tvorbe konečného automatu prechádzaný celý abstraktný syntaktický strom a pre každý podstrom (odpovedajúci podvýrazu) je vytvorený čiastkový konečný automat. Súbor s definíciou vlastností môže obsahovať aj viac vlastností, v tom prípade sa vytvorí pre každú vlastnosť jeden konečný automat.

```
{
  'final_states': {'{1,5}', '{1,3}'},
  'initial_state': '{0,2,4}',
  'input_symbols': {'b', 'a'},
  'states': {'{}', '{0,2,4}', '{1,5}', '{1,3}'},
  'transitions': {'{0,2,4}': {'a': '{1,3}', 'b': '{1,5}'},
                  '{1,3}': {'a': '{}', 'b': '{}'},
                  '{1,5}': {'a': '{}', 'b': '{}'},
                  '{}': {'a': '{}', 'b': '{}'}}
}
```

Výpis 6.4: Príklad reprezentácie determinizovaného konečného automatu pre regulárny výraz $a|b$

V príklade reprezentácie determinizovaného konečného automatu 6.4 značí *final_states* množinu koncových stavov, *initial_state* začiatočný stav, *input_symbols* množinu vstupných symbolov, *states* množinu stavov a slovník *transitions* značí množinu prechodov konečného automatu. Nasledujúci stav v konečnom automate je určený zo slovníka *transitions*, kde sa identifikátor aktuálneho stavu v konečnom automate použije ako index. Výsledkom je slovník prechodov pre aktuálny stav. Použitím identifikátora aktuálnej udalosti ako index je získaný nasledujúci stav.

Vstupná abeceda výsledného konečného automatu pozostáva z identifikátorov udalostí definovaných v sekcii events. Pri príchode udalosti sa nástroj pokúša vykonať prechod v každom konečnom automate.



Obr. 6.4: Sekvenčný diagram znázorňujúci spracovanie vstupu a vytváranie konečných automatov. Trieda *CLI* iniciuje vytvorenie konfiguračného objektu *conf*. Objekt *conf* obsahuje vlastnosti (properties) v nespracovanej forme. Triedy *PropertyParser* a *AutomatonBuilder* spravujú vytvorenie abstraktného syntaktického stromu, resp. konečných automatov pre každú definovanú vlastnosť.

6.2.3 Spracovanie sekcie events

Udalosti v sekcií *events* sú definované ako dvojice $id : regex$, kde id je identifikátor udalosti a $regex$ je regulárny výraz aplikovaný na riadky log súboru. Identifikátor id je malé, alebo veľké písmeno anglickej abecedy.

Regulárny výraz v definícií udalosti je reťazec znakov interpretovaný ako Python regulárny výraz. V definícií udalosti je teda možné použiť všetky špeciálne znaky popísané v [1]. Použitie znaku '\' pred špeciálnym znakom značí doslovný výskyt špecialneho znaku v reťazci.

Znaky okrúhlych zátvoriek '(' a ')' sú špeciálne znaky používané na zachytenie akéhokoľvek regulárneho výrazu medzi týmito zátvorkami. K takto zachyteným dátam je možné sa následne dostať celočíselnými indexami. Index 1 značí obsah prvého výskytu zátvoriek, index 2 obsah druhého výskytu zátvoriek, atď.. Táto vlastnosť je veľmi dobre využiteľná pri definovaní obmedzení nad parametrami popísaných v kapitole 6.2.4.

```
b: bar\((.+),(.+)\)
```

Výpis 6.5: Príklad predpisu udalosti

V 6.5 je príklad predpisu pre zachytenie volania funkcie $bar()$ s dvoma parametrami. Okrúhle zátvorky zvýraznené farbami sú použité na zachytenie dát. Červené zátvorky zachycujú prvý parameter volanej funkcie $bar()$ a zelené zachycujú druhý parameter. Ak by sa v log súbore počas monitorovania vyskytol reťazec $bar(7, 9)$, tak v červených zátvorkách by pod indexom 1 bola hodnota '7' a v zelených pod indexom 2 hodnota '9'.

```
{
 'event_id': 'b',
 'log_file': 'tests/logs/test_input',
 'log_line': bar(7,9),
 'log_lineno': 1,
 'values': ('7', '9')
}
```

Výpis 6.6: Príklad objektu reprezentujúceho udalosť

V 6.6 je príklad reprezentácie udalosti pre výskyt reťazca $bar(7, 9)$ s predpisom pre udalosť uvedenú v 6.5. V tomto objekte značí *event_id* identifikátor udalosti, *log_file* cestu k log súboru z ktorého udalosť pochádza, *log_line* celý riadok z log súboru v ktorom sa daná udalosť vyskytla, *log_lineno* číslo riadku a vo *values* sú uložené hodnoty parametrov v poradí v akom sa vyskytli v udalosti.

6.2.4 Spracovanie sekcie constraints

Obmedzeniami definovanými v sekcii *constraints* sú vyjadrené parametrické závislosti medzi udalosťami. Sú definované ako zoznam výrazov. Zoznam je v YAML formáte definovaný ako postupnosť riadkov začínajucich znakom '-' a medzerou. Obmedzenie je teda v tvare: '– *expression*'. Vo výraze *expression* môžu byť prítomné nasledujúce operátory:

- ('=') rovnosť,
- ('<') menšie ako,

- ('<=') menšie, alebo rovno,
- ('>') väčšie ako,
- ('>=') väčšie, alebo rovno.

Rovnosť je jediný n-árny operátor. Je možné definovať rovnosť n parametrov v tvare $id.number = id.number$ $\{= id.number\}$, kde id je identifikátor udalosti definovanej v sekcií *events* a *number* je celé číslo značiace poradie parametru v udalosti s identifikátorom id. V 6.7 je znázornenie reprezentácie obmedzenia definovaného výrazom $a.2 = c.1 = d.1$. Počas monitorovania je potom umožnené v konečných automatoch vykonávať len tie prechody, ktoré spĺňajú definované rovnosti. V príklade 6.7 sa konkrétne musia rovnať druhý parameter udalosti a, prvý parameter udalosti c a prvý parameter udalosti c.

Rovnosť parametru s konštantou je možné definovať priamo v sekcií *events*, zadaním žiadanej hodnoty parametru priamo do regulárneho výrazu značiaceho udalosť.

```
{
  'events': {'a': '2', 'c': '1', 'd': '1'},
  'op': '='
}
```

Výpis 6.7: Príklad reprezentácie obmedzenia $a.2 = c.1 = d.1$

Pri obmedzeniach obsahujúcich operátory '<', '<=', '>', alebo '>=' je podporovaná len definícia s dvoma operandami. Operand môže byť typu *parameter*, alebo *constant*. V prípade, že ide o typ *parameter*, tak sa hodnota hľadá v aktuálnej udalosti, alebo v minulých výskytoch udalostí. Ak ide o typ *constant*, tak je hodnota pre porovnanie prístupná hneď.

V 6.8 je uvedený príklad reprezentácie pre obmedzenie špecifikované výrazom $b.1 < 42$. Monitorovací algoritmus bude v tomto prípade očakávať, že prvý parameter udalosti b bude menší ako 42. Ak je konštantou reťazec nekonvertovateľný na číslo, tak nástroj končí so syntaktickou chybou.

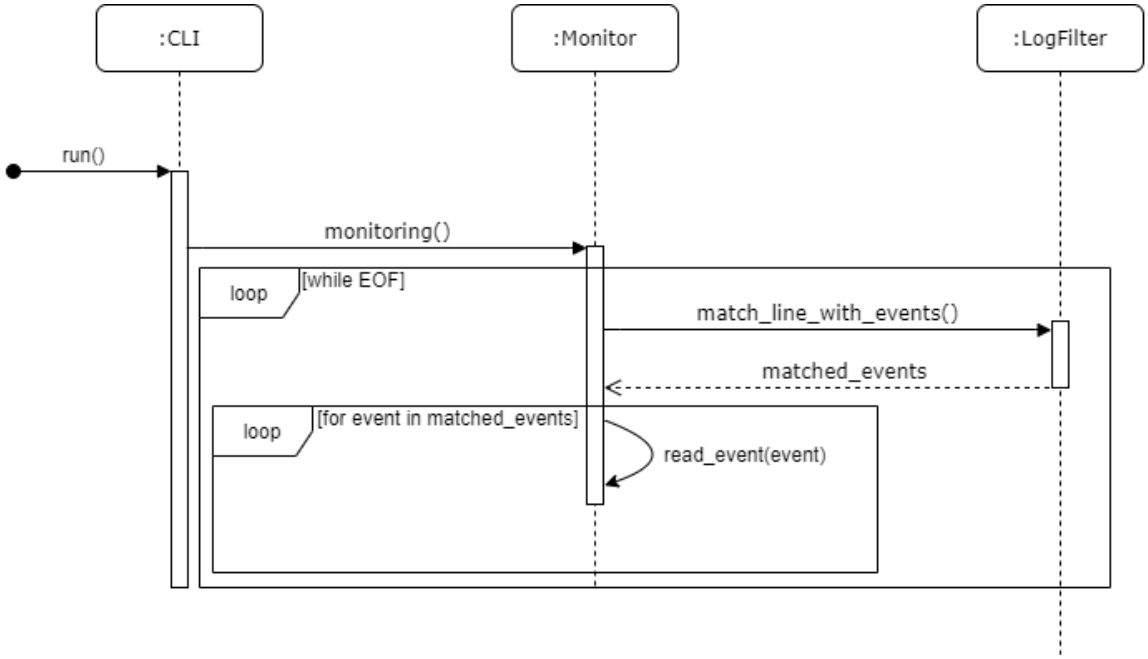
Ide vyjadriť porovnanie dvoch hodnôt parametrov, napríklad $a.1 <= b.2$ (prvý parameter udalosti a musí byť menší, alebo rovný druhému parametru udalosti b). V prípade, že je zadaných viac obmedzení, tak monitorovací algoritmus očakáva, že všetky obmedzenia musia byť splnené.

```
{
  'left': {'event': 'b', 'position': '2', 'type': 'parameter'},
  'right': {'type': 'constant', 'value': 42.0},
  'op': '<'
}
```

Výpis 6.8: Príklad reprezentácie obmedzenia $b.1 < 42$

6.3 Monitorovací algoritmus

Po spracovaní vstupu a vytvorení potrebných dátových štruktúr pre vlastnosti, udalosti a obmedzenia, je spustený monitor. Ten od filtru prijíma spracované udalosti a reaguje na ne. Filter sekvenčne prechádza log súbor a aplikuje naň regulárne výrazy definované v sekcií *events*. Jeden riadok môže obsahovať viac udalostí.



Obr. 6.5: Sekvenčný diagram znázorňujúci spracovanie príchodzích udalostí. Trieda *CLI* spúšta monitorovanie metódou triedy *Monitor*. Monitor sa následne dotazuje metódou z triedy *LogFilter* na prichádzajúce udalosti. Jeden riadok log súboru môže obsahovať viac udalostí. Každá udalosť, ktorá sa vyskytla v riadku je spracovaná metódou *read_event()*.

Na obrázku 6.5 je sekvenčným diagramom popísaný priebeh získania a spracovania udalostí z log súboru. Po iniciovaní monitorovania prechádza filter log súbor až kým nedojde na koniec daného súboru. Získané udalosti z jedného riadku následne monitor sekvenčne prechádza a ďalej na ne reaguje. Na tento diagram nadväzuje diagram 6.7, kde je znázornený mechanizmus vyhodnocovania vlastností.

Každá vlastnosť zadaná v súbore z vlastnosťami je v monitore reprezentovaná konečným automatom. Tieto konečné automaty prijímajú identifikátory udalostí a vykonávajú prechody. Ak sa na konci verifikácie konečný automat nachádza v koncovom stave, tak to znamená, že odpovedajúca vlastnosť je splnená.

```
{
  'au_id': 'automaton_2',
  'automaton': <automata>,
  'current_state': '{3,4}',
  'events': [<event>, <event>],
  'idle': 0,
  'last_transition': 1590587811.6806173,
  'node_index': 2,
  'parameter_map': {'a': ('8',), 'b': ('4','5')},
  'property_id': 'p1'
}
```

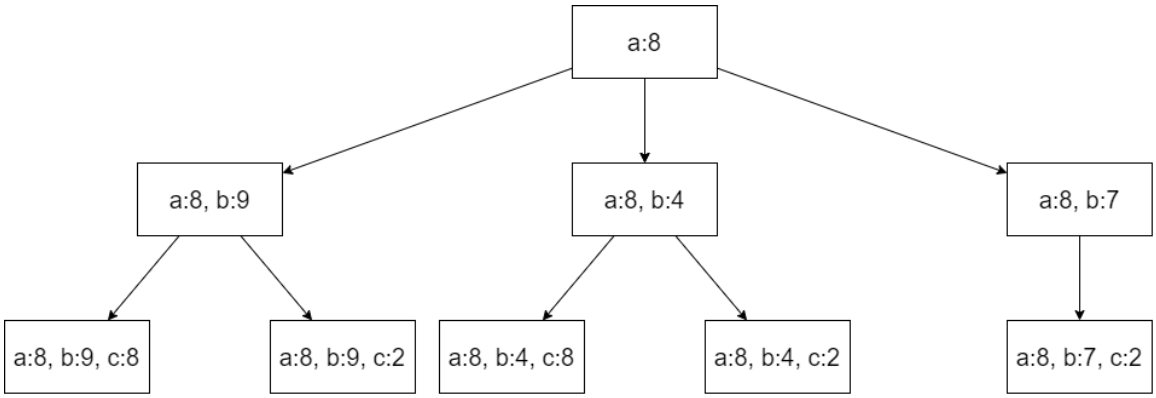
Výpis 6.9: Príklad objektu reprezentujúceho inštanciu konečného automatu

Po príchode udalosti monitor overí pre každý konečný automat, či existuje prechod zo začiatočného stavu. Ak existuje, monitor vytvorí novú inštanciu konečného automatu. Táto inštancia obsahujé dáta o aktuálnom stave a dáta potrebné pre vyhodnocovanie obmedzení. Ďalšie prechody sú vykonávané v týchto inštanciách. Môže existovať viac inštancií pre jednu vlastnosť.

Príklad 6.9 uvádza reprezentáciu inštancie konečného automatu. V tomto objekte značí *au_id* identifikátor inštancie, *automaton* konečný automat prislúchajúci danej inštancií, *current_state* aktuálny stav konečného automatu v ktorom sa inštancia nachádza, *events* zoznam udalostí uchovávaných pre záverečný report, *last_transition* čas posledne vykonaného prechodu v inštancií, *node_index* index do stromu konfigurácií, *parameter_map* slovník uchovávajúci hodnoty všetkých vyskytnutých parametrov v minulosti, *property_id* identifikátor odpovedajúcej vlastnosti. Položka *idle* udáva počet udalostí, počas ktorých bola inštancia neaktívna (nebol vykonaný prechd). Hodnoty *idle* a *last_transition* sú zohľadňované pri uvoľňovaní nepoužívaných prostriedkov. O uvoľňovaní neaktívnych inštancií je viac informácií v kapitole 6.6.

Hodnoty v slovníku *parameter_map* sú usporiadané n-tice, Hodnoty v n-tico sú usporadané podľa poradia výskytu parametrov v udalosti. Konkrétne, inštancia v príkade 6.9 vykonala dva prechody s udalosťami a a b. Jediný parameter udalosti a mal hodnotu 8. Prvý parameter udalosti b mal hodnotu 4 a druhý 5.

S príchodom každej udalosti monitor sekvenčne prechádza jednotlivné inštancie konečných automatov a aktualizuje ich aktuálny stav v konečnom automate. Tiež je aktualizovaný slovník minulých hodnôt parametrov podľa udalostí, ktoré nastali a ich hodnôt parametrov.



Obr. 6.6: Príklad stromu konfigurácií pre vlastnosť *abc* bez obmedzení. Pre jednoduchosť je v uzloch uvedená len položka *parameter_map*, v ktorej sú uchovávané hodnoty parametrov. Udalosti a, b a c majú každá jeden celočíselný parameter a sú definované v tvare: $a|b|c\ num$, kde num je celé číslo. Konfiguračný strom na obrázku odpovedá výskytu udalostí v nasledujúcom poradí: a 8, b 9, b 4, c 8, b 7, c 2.

Pre overenie parametrických vlastostí je však potrebné kontrolovať nielen aktuálny stav v konečnom automate, ale aj všetky stavy a hodnoty parametrov vyskytnutých v minulosti. Majme vlastnosť *abc* a sekvenciu udalostí s jedným parametrom v nasledujúcom poradí: a 1, b 1, c 1, b 2. Po prvej udalosti a 1 je vytvorená inštancia konečného automatu a vykonaný prechod cez identifikátor udalosti a. Inštancia je teraz v stave konečného automatu kedy očakáva príchod udalosti s identifikátorom b. Podobný posun nastane pri udalosti b 1. Príchodom udalosti c 1 sa inštancia dostane do koncového stavu kedy neočakáva žiadnu

ďalšiu udalosť. Keď nastane udalosť b 2, v inštancií nie je k dispozícií prechod, pretože sa už nachádza v koncovom stave. Ide o už raz vyskytnutú udalosť b, ale s hodnotou parametru, ktorá sa ešte v udalosti b neobjavila. Preto je na ňu nahliadané ako na unikátnu. Táto udalosť vytvára novú sekvenciu: a 1, b 2. Keďže žiadne ďalšie udalosti nie sú k dispozícií, definovaná vlastnosť abc nie je pre sekvenciu a 1, b 2 splnená.

Výsledný nástroj by mal vedieť overovať aj sekvencie odlišujúce sa hodnotami parametrov. Jedna z možností ako toto umožniť je pri výskyte každej udalosti overovať prechod z každého v minulosti navštíveného stavu konečného automatu. Ak je z nejakého minulého stavu prechod k dispozícií, tak sa vytvorí nová inštancia a na nej sa vykoná daný prechod. Preto je potrebné si uchovávať všetky stavy konečných automatov navštívených v minulosti a hodnoty ich parametrov.

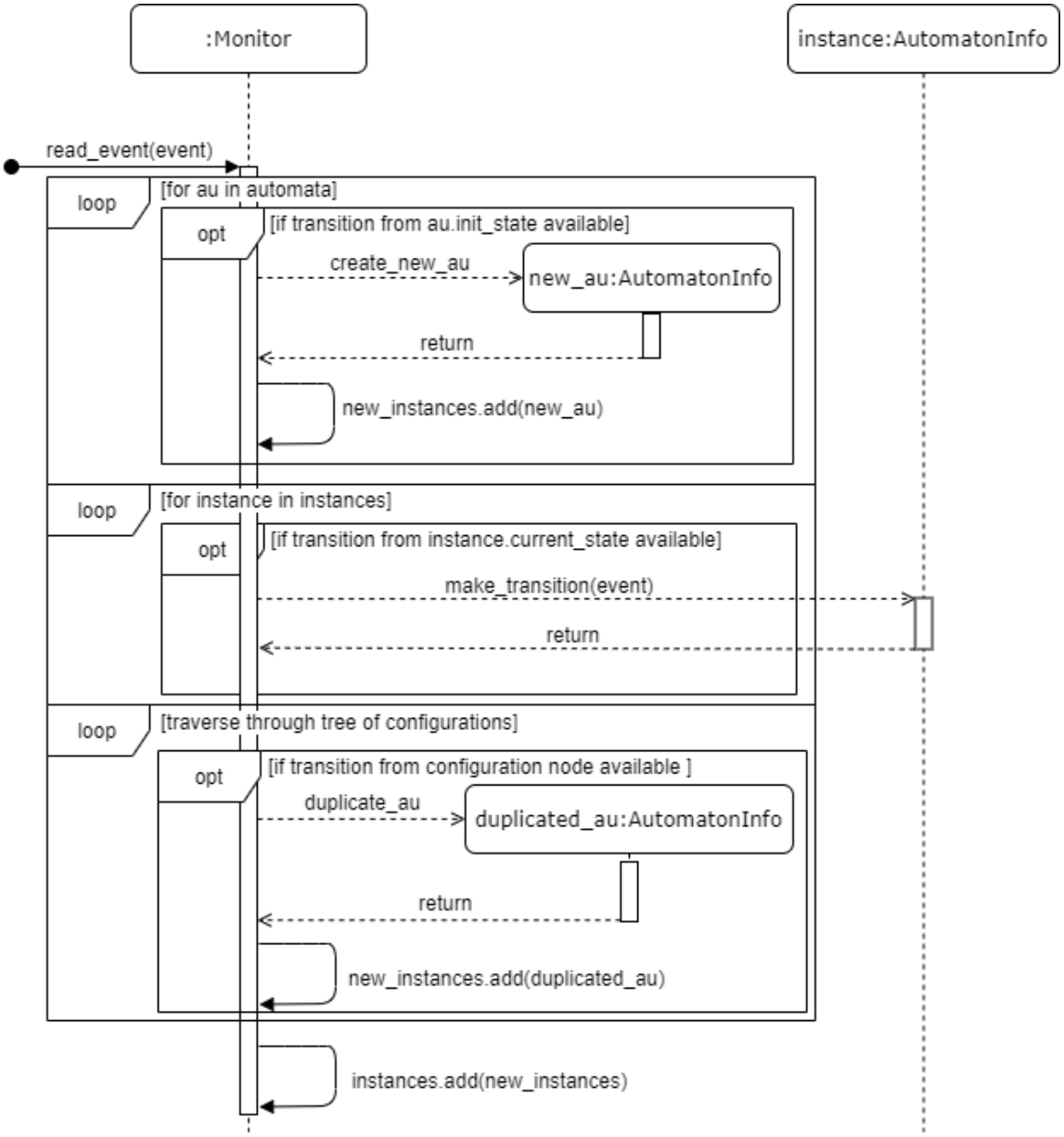
Pre tento účel slúži objekt konfigurácia. V konfigurácií je uložený stav konečného automatu a mapovanie parametrov na hodnoty. Konfigurácie sú usporiadané v stromovej štruktúre. Ak bol niekedy v minulosti vykonaný prechod z jedného do druhého stavu v konečnom automate, v konfiguračnom strome bude tento prechod zaznamenaný ako vzťah rodič-potomok.

Na obrázku 6.6 je znázornený konfiguračný strom pre vlastnosť abc s udalosťami s jedným parametrom. Uzly predstavujú konfiguráciu. Na obrázku sú do nich vpísané hodnoty parametrov, ktoré boli zaznamenané do daného stavu. Uzol s hodnotami $a : 8$ je koreňový uzol a značí konfiguráciu po prijatí udalosti a s hodnotou parametra 8. Nasledujúca udalosť b 9 posunie existujúcu inštanciu o jeden stav a vytvorí sa nová konfigurácia odpovedajúca novému stavu (uzol s hodnotami $a : 8$ $b : 9$). Pri ďalšej udalosti $b4$ je možný vykonať prechod s uzlu $a : 8$. Preto je vytvorená nová inštancia obsahujúca

Monitorovací algoritmus s príchodom každej udalosti prechádza konfiguračný strom a pokúša sa vykonať prechod z každého konfiguračného uzlu, ktorý nie je listový (s ohľadom na parametrické obmedzenia popísané v kapitole 6.3.1). Všetky cesty v grafe k listovým uzlom predstavujú sekvencie udalostí. Každá z týchto sekvencií je reprezentovaná inštanciou konečného automatu. Konkrétne, sú to sekvencie:

1. a 8, b 9, c 8
2. a 8, b 9, c 2
3. a 8, b 4, c 8
4. a 8, b 4, c 2
5. a 8, b 7, c 2

Pre každú prvú udalosť v sekvencií je vytvorený samostatný strom konfigurácií. Každej definovanej vlastnosti môže teda odpovedať niekoľko týchto stromov.



Obr. 6.7: Sekvenčný diagram popisujúci monitorovací algoritmus. Nadväzuje na diagram 6.5.

Na obrázku 6.7 je znázornený monitorovací algoritmus sekvenčným diagramom. Akcie v diagrame sú vykonávané pre každú udalosť (*event*) z log súboru. Algoritmus pozostáva z troch hlavných cyklov. Prvý cyklus prechádza všetky konečné automaty odpovedajúce definovaným vlastnostiam. Ak sa dá vykonať prechod zo začiatočného stavu nejakého automatu s aktuálnou udalosťou, tak monitor vytvorí novú inštanciu *new_au* a na tejto novej inštancií prechod realizuje.

Druhý cyklus prechádza všetky inštancie. Ak ide v nejakej inštancií vykonať prechod z aktuálneho stavu, tak ho vykoná a aktualizuje dáta v danej inštancií.

Tretí cyklus prechádza všetkými stromami konfigurácií. Ak je možné z nejakého konfiguračného uzlu vykonať prechod, tak monitor vytvorí novú inštanciu. Táto nová inštancia

bude obsahovať sekvenciu udalostí získanú z cesty v grafe k danému uzlu. V novej inštancií je následne vykonaný prechod podľa aktuálnej udalosti.

Ak po nejakom z prechodov sa nejaká inštancia dostane do koncového stavu, zo zoznamu inštancií je vymazaná. Ak išlo o inštanciu odpovedajúcu negatívnej vlastnosti (definovanú v sekcií *badproperties*), tak je sekvencia udalostí danej inštancie pridaná do finálneho reportu.

Ak na konci verifikácie zostanú nejaké inštancie odpovedajúce pozitívnym vlastnostiam (definované v sekcií *properties*) v inom ako koncovom stave, tak sú sekvencie udalostí týchto inštancií pridané do záverečného reportu a označené ako nesplnené.

6.3.1 Overovanie obmedzení nad parametrami

Pri výskyte nejakej udalosti sú overované prechody v inštanciách konečných automatov. Ak je nejaký prechod k dispozícií, tak sú následne overené aj obmedzenia nad parametrami. Tieto obmedzenia sú definované v súbore s definíciou vlastností v sekcií *constraints*. Ich spracovanie bolo popísané v kapitole 6.2.4. V prípade, že nejaké obmedzenie nie je splnené, tak sa prechod v konečnom automate nevykoná.

Ako bolo spomenuté v kapitole 6.2.4, sú dva typy obmedzení: rovnosť a porovnanie. Rovnosť je n-árna operácia s minimom dvoch operandov. Porovnanie je binárna operácia. Umožňuje porovnať hodnotu parametru a konštanty, alebo hodnôt dvoch parametrov. Porovnanie podporuje operátory: $<$, $<=$, $>$ a $>=$.

Hodnoty parametrov sú získané zo slovníka hodnôt, ktorý má každá inštancia. Slovník môže obsahovať napríklad tieto hodnoty:

```
{
  'a': ('8',),
  'b': ('4','5')
}
```

ak by sa v definícií obmedzení vyskytol výraz $b.2$, odpovedal by hodnote 5. Ak hodnota parametru v tomto slovníku nie je nájdená, hľadá sa v aktuálnej udalosti. Ak pri vyhodnocovaní nie je nejaká hodnota nájdená ani v aktuálnej udalosti a ani v slovníku hodnôt, tak sa predpokladá, že sa vyskytne v nejakej udalosti v budúcnosti. To znamená, že obmedzenie môže byť vyhodnotené ako nesplnené len v prípade, že boli definované všetky parametre v definícií daného obmedzenia.

Majme definovanú pozitívnu (musí byť splnená) vlastnosť ab. Udalosti a a b majú každá jeden celočíselný parameter a sú definované v tvare: $a|b\ num$, kde num je celé číslo. Nech je definované obmedzenie $a.1 = b.1$ a sekvencia udalostí: a 1, b 2. Po príchode udalosti a 1 je vytvorená inštancia automatu so slovníkom hodnôt v nasledujúcom stave:

```
{
  'a': ('1',)
}
```

Po príchode udalosti b 2 monitorovací algoritmus overí existenciu prechodu v odpovedajúcom konečnom automate. Prechod je k dispozícií, ale obmedzenie $a.1 = b.1$ nie je splnené. Hodnota podvýrazu $a.1$ bola získaná zo slovníka hodnôt a hodnota $b.1$ bola získaná z aktuálnej udalosti b 2. Keďže obmedzenie nebolo splnené v inštancií sa prechod nevykoná a vlastnosť je nesplnená. Porovnanie hodnôt inými operátormi je realizované obdobne. Ak je definovaných viac obmedzení, na umožnenie prechodu musia byť vyhodnotené všetky obmedzenia ako splnené.

Nástroj *plogchecker* umožňuje v definícií vlastností aj opakovanie udalostí. Majme definíciu vlastnosti *aa*. Udalosť *a* má jeden celočíselný parameter a je definovaná v tvare: *a num*, kde *num* je celé číslo. Majme sekvenciu udalostí: *a* 1, *a* 2. Po príchode udalosti *a* 1 je vytvorená nová inštancia konečného automatu. Aktuálny stav inštancie má jediný prechod do koncového stavu cez udalosť *a*. Nasledujúca udalosť *a* 2 má síce identifikátor, ktorý jediná inštancia očakáva, má ale inú hodnotu parametru. Preto sa prechod v inštancií nevykoná. Pri výskyte rovnakých udalostí v definícií vlastnosti sa očakáva, že rovnaké budú aj všetky hodnoty ich parametrov. To isté platí aj pri operátoroch iterácie.

Pre zamedzenie vytvárania nadbytočných inštancií z vlastností, ktoré obsahujú opakované udalosti, je vytvorený globálny slovník hodnôt. Tu sú uložené všetky použité hodnoty parametrov naprieč inštanciami. Ak je po príchode udalosti k dispozícií prechod zo začiatočného stavu nejakého konečného automatu, ale daná kombinácia parametrov bola už raz použitá, tak nová inštancia nebude vytvorená.

6.4 Finálny report

Výsledný report má za úlohu vyjadriť ktoré vlastnosti boli splnené a ktoré porušené. Je vo formáte JSON a obsahuje nasledovné položky:

- zoznam vlastností rozdelený na pozitívne (*properties*) a negatívne (*badproperties*),
- vlastnosť obsahuje:
  - názov vlastnosti,
  - definíciu vlastnosti (regulárny výraz),
- sekvenciu udalostí, ktoré vlastnosť porušili (je prítomná len v prípade, že bola vlastnosť porušená)
- udalosť obsahuje:
  - identifikátor udalosti,
  - cesta k log súboru v ktorom sa udalosť vyskytla,
  - číslo riadku log súboru v ktorom sa udalosť vyskytla,
  - presné znenie riadku log súboru.

Formát výsledného reportu špecifikovaný v JSON Schema je v prílohe A.

6.5 Prúdové spracovanie

Výstupom nástroja *logchecker* je report o porušených vlastnostiach. Tento report je však vygenerovaný až po skončení behu nástroja. Užívateľ sa teda dostane k výsledkom až po ukončení verifikácie. Existuje však možnosť informovať užívateľa o aktuálnom stave verifikácie aspoň čiastočne.

Pozitívne vlastnosti sú považované za splnené pokiaľ sa ich odpovedajúci konečný automat nachádza v koncovom stave. Pokiaľ sa uprostred verifikácie nejaký konečný automat nenachádza v koncovom stave, nasledujúca postupnosť udalostí môže tento automat dostať do koncového stavu. Nie je teda možné s určitosťou rozhodnúť, že daná pozitívna vlastnosť je splnená, alebo porušená pokiaľ nie je verifikácia na konci sekvencie udalostí.

Pokiaľ sa konečný automat prislúchajúci k negatívnej vlastnosti nachádza v koncovom stave, tak sa vlastnosť považuje za porušenú. Ak sa nejaký konečný automat reprezentujúci negatívnu vlastnosť dostane uprostred verifikácie do koncového stavu, tak je možné túto vlastnosť s istotou prehlásiť za porušenú. Počas behu nástroja je teda možné informovať užívateľa o porušených negatívnych vlastnostiach.

```
Bad property "de" with id "p2" was violated!

   Sequence of events which caused violation:

     3. foo(9)
         event id = d
         log file is "tests/logs/log_file"

     5. bar(7)
         event id = e
         log file is "tests/logs/log_file"
```

Výpis 6.10: Príklad prúdového výstupu počas behu nástroja plogchecker

Na 6.10 je znázornený príklad priebežného výpisu o porušenej vlastnosti v textovom formáte. Ide o porušenú vlastnosť `p2` definovanú ako *de*. Vo výpise sú presné znenia riadkov logu s odpovedajúcimi číslami riadkov, identifikátormi udalostí a cestou k log súboru. Je možné priebežne vypisovať porušené negatívne vlastnosti aj vo formáte JSON.

6.6 Uvoľňovanie prostriedkov

Počas monitorovania udalostí sa môže stať, že pri vysokom počte vyskytnutých parametrov nástroju dojde dostupná pamäť. Preto je potrebné priebežne uvoľňovať dlho nepoužité prostriedky pre oddialenie pádu nástroja. V súčasnom stave sú navrhnuté dva spôsoby uvoľňovania prostriedkov.

6.6.1 Sekvenčné uvoľňovanie

Pri vyhodnocovaní udalostí monitorovací algoritmus prechádza sekvenčne cez každú inštanciu konečných automatov a snaží sa vykonať prechod. Ak sa nepodarí vykonať v danej inštancií prechod, zvýši sa hodnota položky *idle* o jedna. Hodnota tejto položky značí počet udalosti po ktorých bola daná inštancia nečinná (nebol v nej vykonaný prechod). Táto hodnota je smerodajná pri výbere nečinných inštancií. Ak hodnota *idle* prekročí nejakú dopredu stanovenú hodnotu (zadanú parametrom príkazového riadku), tak odpovedajúca inštancia bude zmazaná zo zoznamu inštancií. Každé zmazanie je sprevádzané správou o zmazaní na štandartný výstup.

6.6.2 Paralelné uvoľňovanie

Sekvenčné uvoľňovanie je vhodné pre aktuálnu implementáciu nástroja *plogchecker*, pretože pri vyhodnocovaní nie je využitý paralelizmus a zoznam inštancií je prechádzaný za kaž-

dých okolností sekvenčne. Tento spôsob vyhodnocovania umožňuje jednoducho udržiavať hodnotu udávajúcu dobu nečinnosti danej inštancie.

Pri eventuálnom rozšírení nástroja o paralelizmus je však predchádzajúci spôsob neefektívny. Preto je navrhnutý a implementovaný aj spôsob uvoľňovania vhodný pre paralelnú implementáciu. Princíp tohto uvoľňovania je v uchovávaní času kedy bola daná inštancia naposledy aktívna. Pri vykonaní prechodu v danej inštancií je do položky *last_transition* uložený aktuálny čas. Po prekročení určitého počtu inštancií (počet je daný parametrom príkazového riadku) sa spustí uvoľňovanie neaktívnych inštancií. Pri uvoľňovaní sa zistia najmenej aktívne inštancie podľa ich hodnôt *last_transition*. Inštancie s najstarším časom posledného prechodu sú zmazané. Počet zmazaných inštancií je daný parametrom príkazového riadku.

6.7 Časová zložitosť

Po spustení nástroja sú z definícií vlastností vytvorené odpovedajúce šablóny konečných automatov. Najprv sú prevedené regulárne výrazy na nedeterministické konečné automaty. Asymptotická zložitosť tejto operácie je $O(n)$, kde n je počet stavov výsledného nedeterministického automatu. Stavy nedeterministického automatu sú vytvárané sekvenčným prechádzaním regulárneho výrazu. Každá operácia v regulárnom výraze vytvára maximálne štyry nové stavy. Keďže je počet stavov pre jednu operáciu obmedzený na maximálne štyry, tak počet výsledných stavov je určený len dĺžkou regulárneho výrazu. Asymptotická zložitosť tohto prevodu je teda $O(m)$, kde m je dĺžka regulárneho výrazu.

Následne sú všetky nedeterministické konečné automaty prevedené na deterministické. Stavy deterministického automatu pozostávajú z množín stavov pôvodného nedeterministického automatu. V najhoršom prípade je teda potrebné vytvoriť 2^n stavov, kde n je počet stavov pôvodného nedeterministického automatu. Počet stavov nedeterministického automatu sa priamo odvíja od dĺžky pôvodného regulárneho výrazu. Preto sa dá zložitosť prevodu na deterministický konečný automat vyjadriť ako $O(2^m)$, kde m je dĺžka vstupného regulárneho výrazu.

Celková zložitosť prevodu regulárneho výrazu na deterministický konečný automat je $O(m + 2^m)$. Hodnoty 2^m sú nepochybne vyššie ako hodnoty m, preto sa celková zložitosť prevodu dá zapísať ako $O(2^m)$. Počet konečných automatov je určený počtom definovaných vlastností. Regulárny výraz sa teda bude prevádzať na deterministický automat toľko krát, koľko je vlastností. Celková asymptotická zložitosť prevodu všetkých regulárnych výrazov na deterministické konečné automaty je teda $O(k.2^m)$, kde k je počet definovaných vlastností a m je dĺžka regulárneho výrazu.

Po vytvorení konečných automatov je spustené monitorovanie udalostí. Pre každú udalosť sú vykonávané tri akcie. Priebeh reakcie na jednotlivé udalosti je znázornený na obrázku 6.7. Po príchode udalosti je najprv overené, či sa nedá vykonať prechod zo začiatočného stavu nejakého konečného automatu. Algoritmus iteruje cez všetky konečné automaty a overuje, či je možné vykonať validný prechod. Pri overovaní prechodu sú kontrolované aj obmedzenia pre porovnanie parametrov udalosti s konštantou. Priebeh tohto overenia má celkovú zložitosť $O(k.c)$, kde k je počet definovaných vlastností a c je počet definovaných obmedzení.

Ďalšia fáza overovania udalosti je kontrolovanie prechodov v inštanciách konečných automatov. Algoritmus iteruje cez všetky inštancie. Počet inštancií závisí od počtu výskytov nových hodnôt parametrov v udalostiach. Pri overovaní prechodov sú kontrolované aj operácie rovnosti u obmedzení. Rovnosť je n-árna operácia, môže mať teda n parametrov. Pre

každú inštanciu je teda overované každé obmedzenie, ktoré môže mať niekoľko pozičných parametrov. Priebeh tohto overenia má celkovú zložitosť $O(j.c.l)$, kde j je počet výskytov unikátnych hodnôt parametrov, c je počet definovaných obmedzení a l počet parametrov operácie rovnosti.

Posledná fáza pri overovaní udalosti je prechádzanie stromami konfigurácií a testovanie prechodu z každého uzlu. Počet uzlov sa odvíja množstva doteraz vykonaných prechodov i. Pre každý uzol je potrebné overiť existenciu validného prechodu, preto je potrebné prechádzať všetky definované obmedzenia a ich parametre. Celková zložitosť tejto fázy je $O(i.c.l)$, kde i je počet vykonaných prechodov, c počet definovaných obmedzení a l počet parametrov operácie rovnosti.

Tieto tri fázy sú vykonávané pre každú jednu udalosť, preto celková zložitosť monitorovacieho algoritmu je $O(p).(O(k.c)+O(j.c.l)+O(i.c.l))$, kde p je počet udalostí. Celková asymptotická zložitosť behu nástroja *plogchecker* je $O(k.2^m)+O(p).(O(k.c)+O(j.c.l)+O(i.c.l))$.

6.8 Priestorová zložitosť

Nástroj *plogchecker* potrebuje pri výpočte niekoľko datových štruktúr. Počet a veľkosť deterministických automatov sú určené počtom vlastností a dĺžkou zadaných regulárnych výrazov. V najhoršom prípade môže mať výsledný deterministický konečný automat až 2^n stavov, kde n je dĺžka pôvodného regulárneho výrazu. Priestorová zložitosť konečných automatov je $O(k.2^n)$, kde k je počet definovaných udalostí a n je dĺžka regulárneho výrazu.

Ďalšou dátovou štruktúrou sú inštancie konečných automatov. Ich množstvo závisí od počtu výskytov nových hodnôt parametrov v sekvencií udalostí. Priestorová zložitosť inštancií konečných automatov je $O(j)$, kde j je počet výskytov unikátnych hodnôt v sekvencií udalostí.

Pri overovaní udalostí je potrebné uchovávať všetky minulé konfigurácie konečných automatov a ich mapovanie parametrov na hodnoty. Množstvo týchto konfiguračných uzlov je dané počtom vykonaných validných prechodov. Priestorová zložitosť konfigurácií je $O(i)$, kde i je počet validných prechodov.

Je potrebné uchovávať aj definované obmedzenia. Výsledná priestorová zložitosť obmedzení je $O(c.l)$, kde c je počet definovaných obmedzení a l je počet parametrov v operácií rovnosti.

Pre správne vyhodnotenie vlastností je potrebné uchovávať všetky udalosti, ktoré nastali a sú súčasťou nejakej sekvencie. Priestorová zložitosť uložených udalostí je $O(p)$, kde p je počet vyskytnutých udalostí.

Celková priestorová zložitosť nástroja *plogchecker* je $O(k.2^n)+O(j)+O(i)+O(c.l)+O(p)$.

6.9 Vyjadrovacia schopnosť jazyka nástroja *plogchecker*

Vlastnosti sú v nástroji *plogchecker* definované regulárnymi výrazmi. Regulárne výrazy patria v Chomského hierarchií medzi jazyky generované regulárnymi gramatikami. Vo výslednom nástroji sa pri každej novej hodnote parametru vytvára nová inštancia konečného automatu. Táto nová inštancia následne prijíma udalosti a vykonáva prechody bez znalosti o ostatných inštanciách.

Nech je regulárny výraz ab definícia sekvencie udalostí s jedným celočíselným parametrom. Udalosti sú definované v tvare: $a|b\ num$, kde num je celé číslo. Nech je definované obmedzenie $a.1 = b.1$. Pri každom novom výskyte parametru udalosti a sa vytvorí nová

inštancia konečného automatu. Táto inštancia bude očakávať udalosť b s rovnakou hodnotou parametru akú mala udalosť a pri prvom prechode inštancie. Príklad sekvencie udalostí spĺňajúcej definovanú vlastnosť je napríklad:

```
a 1
a 2
b 1
a 3
b 2
b 3
```

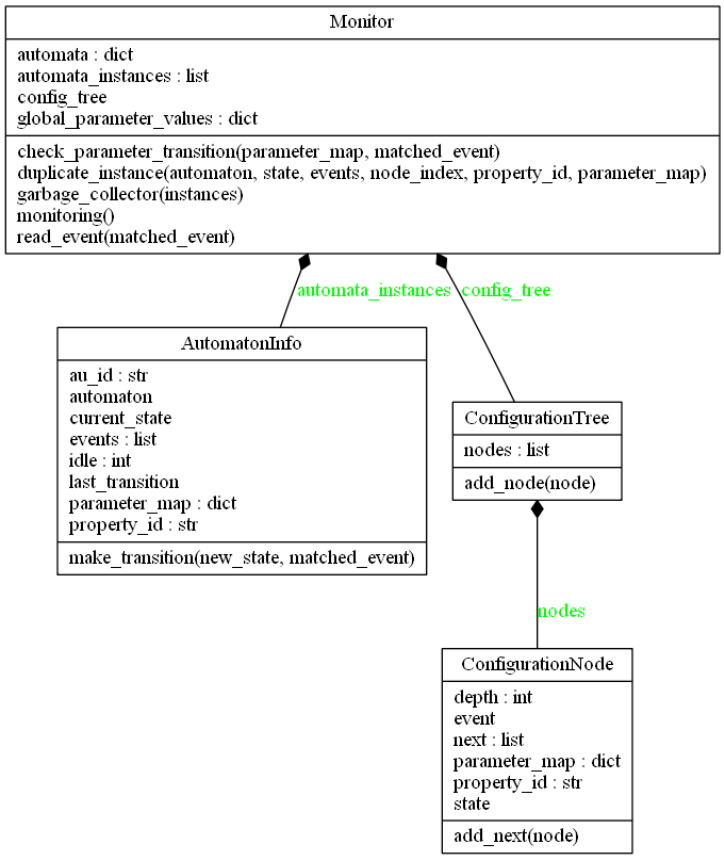
Sekvencie udalostí spĺňajúce definovanú vlastnosť zjavne nie je možné prijať bežným konečným automatom. Vlastnosť ab s obmedzením $a.1 = b.1$ je podobná bezkontextovému jazyku $L = \{a^n b^n; n \in N\}$. Narozdiel od zásobníkového automatu, mechanizmus nástroja *plog-checker* nedokáže zaistiť poradie výskytu udalostí naprieč hodnotami parametrov. Každá inštancia konečného automatu dokáže len prijať sekvenciu udalostí so svojími hodnotami parametrov.

Kapitola 7

Implementačné detaily nástroja plogchecker

V tejto kapitole sú popísané implementačné detaily výsledného nástroja. V nasledujúcich podkapitolách sú priblížené najdôležitejšie moduly výsledného nástroja. V poslednej podkapitole 7.3 je návod na spustenie nástroja *plogchecker*.

7.1 Modul *monitor*



Obr. 7.1: Štruktúra modulu *monitor* znázornená diagramom tried

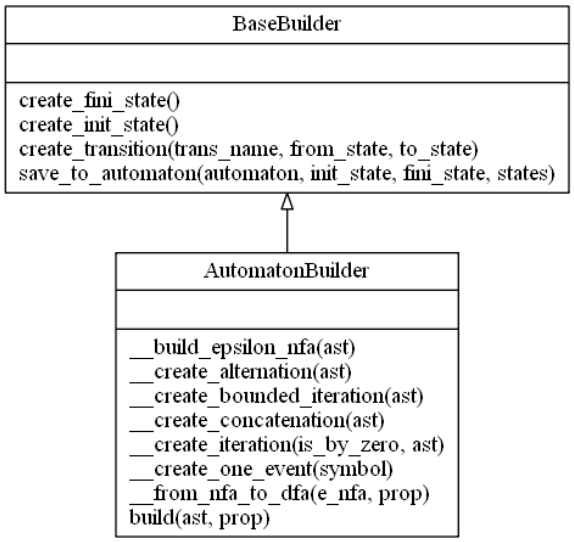
V tomto module je implementovaný monitorovacií algoritmus. Na obrázku 7.2 je znázornená štruktúra modelu diagramom tried. V diagrame sú uvedené najdôležitejšie atribúty a metódy. Monitorovanie je spustené metódou *monitoring* z triedy *Monitor*. Táto metóda iteruje cez prijímané udalosti od filtru a pre každú udalosť je volaná metóda *read_event*. Šablóny konečných automatov sú uložené v slovníku *automata*. Inštancie konečných automatov sú uložené v zozname *automata_instances*. V *config_tree* je uložený strom konfigurácií *ConfigurationTree*. Parametrické obmedzenia sú overované v metóde *check_parameter_transition*. Mapovanie hodnôt na parametre je ukladané v slovníku *global_parameter_values*. Ak sa dá spraviť validný prechod z nejakého uzlu v strome *config_tree*, tak je vytvorená nová inštancia *AutomatonInfo* metódou *duplicate_instance*. Uvoľňovanie prostriedkov je realizované metódou *garbage_collection*.

Inštancie konečných automatov sú reprezentované objektami triedy *AutomatonInfo*. V týchto objektoch značí *au_id* identifikátor inštancie, *automaton* odpovedajúcu šablónu konečného automatu, *current_state* aktuálny stav v ktorom sa inštancia nachádza, *events* zoznam udalostí v poradí v akom nastali, *idle* počet udalostí po ktoré bola inštancia neaktívna, *last_transition* čas kedy bola inštancia naposledy aktívna, *parameter_map* slovník kde je uložené mapovanie parametrov na hodnoty, *property_id* identifikátor odpovedajúcej vlastnosti. Metódou *make_transition* je vykonávaný prechod v inštancií konečného automatu.

ConfigurationTree predstavuje strom konfigurácií. Obsahuje zoznam konfiguračných uzlov *nodes*. Prvky tohto zoznamu sú objekty triedy *ConfigurationNode*. Táto trieda obsahuje jedinú metódu *add_node*, ktorou je pridaný nový uzol do konfiguračného stromu.

Uzly konfiguračného stromu sú objekty triedy *ConfigurationNode*. V týchto objektoch znamená *depth* úroveň v ktorej sa daný uzol nachádza v strome, *event* udalosť po ktorej daný uzol vznikol, *next* zoznam nasledujúcich uzlov, *parameter_map* slovník hodnôt parametrov, *property_id* identifikátor odpovedajúcej vlastnosti, *state* odpovedajúci stav v konečnom automate.

7.2 Modul *au_builder*



Obr. 7.2: Štruktúra modulu *au_builder* znázornená diagramom tried

V module *au_builder* je implementované vytváranie konečných automatov z definície vlastností. Modul obsahuje triedy *BaseBuilder* a *AutomatonBuilder*. V triede *BaseBuilder* sú implementované operácie potrebné pre vytvorenie nedeterministického konečného automatu s ε-prechodmi. Konkrétne *create_fini_state* vytvára konečný stav automatu, *create_init_state* vytvára začiatočný stav automatu a *create_transition* vytvára prechod medzi dvoma stavmi konečného automatu. Metóda *save_to_automaton* slúži na dočasné uloženie automatu odpovedajúceho podstromu abstraktného syntaktického stromu.

Trieda *AutomatonBuilder* obsahuje metódy pre vytvorenie nedeterministických konečných automatov z abstraktného syntaktického stromu. Metóda *build_epsilon_nfa* vytvára nedeterministický konečný automat z abstraktného syntaktického stromu. Výsledný konečný automat je vytvorený spojením čiastkových automatov odpovedajúcich podstromom abstraktného syntaktického stromu. Čiastkové konečné automaty sú vytvárané podľa operátora v abstraktnom syntaktickom strome metódami *create_alternation* (alternácia), *create_bounded_iteration* (obmedzená iterácia), *create_concatenation* (konkatenácia) a *create_iterations* (iterácia). Metóda *create_one_event* vytvára čiastkový konečný automat odpovedajúci jednému symbolu. Metóda *from_nfa_to_dfa* transformuje nedeterministický konečný automat na deterministický. Využíva pri tom funkcie z knižnice Automata [7].

7.3 Spustenie nástroja

Nástroj vyžaduje verziu Python 3.7.3. Balíky potrebné pre beh nástroja sú uvedené v súbore `requirements.txt`. Ich nainštalovanie je možné pomocou správcu balíkov *pip* spustením príkazu:

```
pip install -r requirements.txt
```

7.3.1 Parametre nástroja

Medzi povinné parametre patrí:

- `-p, --properties FILE` - súbor s definíciou vlasností a udalostí vo formáte YAML

Medzi voliteľné parametre patrí:

- `-l, --log FILE` - log súbor. Ak parameter nie je zadaný, nástroj očakáva udalosti na štandartnom vstupe.
- `-r, --report DIR` - adresár kde má byť uložený výsledný report. Ak nie je zadaný, je uložený do koreňového adresára nástroja.
- `-s, --stream [JSON, TEXT]` - zadanie jednej z hodnôt JSON, TEXT zapína prúdové spracovanie. Porušené negatívne vlastnosti sú vypisované priamo na štandartný výstup počas behu nástroja. Formát výstupu môže byť vo formáte JSON, alebo v čítateľnom textovom formáte.
- `-v, --verbose [1, 2, 3]` - ak užívateľ zadá jednu z troch hodnôt, bude spustený *verbose* mód. Po zadaní hodnoty 1 sú na štandartný výstup vypísané riadky log súboru, ktoré neobsahovali
- `-t, --timeout N` - parameter udávajúci dobu nečinnosti inštancie (udávanú v počte ubehnutých udalostí) potrebnú na jej zmazanie pri uvoľňovaní prostriedkov. Ak tento parameter nie je zadaný, uvoľňovanie prostriedkov počas behu nástroja neprebieha.

žiadne udalosti. Po zadaní 2 sú vypísané čísla riadkov log súboru a identifikátory udalostí ktoré daný riadok obsahuje. Po zadaní hodnoty 3 sú vypísané čísla riadkov log súboru a identifikátory udalostí s hodnotami parametrov ktoré daný riadok obsahuje.

- `-t, --timeout N` - parameter udávajúci dobu nečinnosti inštancie (udávanú v počte ubehnutých udalostí) potrebnú na jej zmazanie pri uvoľňovaní prostriedkov. Ak tento parameter nie je zadaný, uvoľňovanie prostriedkov počas behu nástroja neprebieha.
- `-g, --garbage START NUM` - parameter definovaný dvoma hodnotami. Hodnota `START` udáva pri akom počte inštancií bude paralelné uvoľňovanie prostriedkov spustené. Hodnota `NUM` udáva počet najmenej aktívnych inštancii, ktoré budú zmazané.
- `-d, --debug` - ak je tento prepínač zadaný, tak sú na štandartný výstup priebežne vypisované debugovacie informácie

Príklad spustenia nástroja so súborom s vlastnosťami `win_firewall.yml` a log súborom `win_firewall`:

```
python plogchecker.py -p win_firewall.yml -l win_firewall
```

Po dokončení behu bude v koreňovom adresáry nástroja vytvorený súbor `report.json`. V tomto súbore budú znázornené postupnosti udalostí ktoré nevyhovovali špecifikovaným vlastnostiam.

Kapitola 8

Popis testovania nástroja plogchecker

V tejto kapitole je popísané testovanie výsledného nástroja. Funkcionalita bola overená automatickými testami. Výkonnosť bola otestovaná manuálnym spúšťaním nástroja na rôznych vstupoch a sledovaní časovych a pamäťových nárokov.

8.1 Testovanie funkcionality

Funkcionálne testovanie prebiehalo pomocou automatických testov. Automatické testy boli implementované v nástroji pytest [13]. Tento nástroj umožňuje spúšťanie testovacích scenárov, ich vyhodnotenie a oznámenie tých testovacích scenárov ktoré zlyhali. Vstupom každého testovacieho scenára je cesta k súboru s vlastnosťami a cesta k log súboru. Testovacie súbory s vlastnosťami sú uložené v adresáry `tests/properties/`. Testovacie log súbory sú v adresáry `tests/logs/`. Referenčné súbory sú v adresáry `tests/outputs/`. Všetky testovacie súbory majú v názve predponu `test`. Výstup testovacieho scenáru je porovnaný s referenčným výstupným súborom. Ak *pytest* zaznamená, že výstup nástroja je rozdielny s referenčným súborom, testovací scenár označí ako zlyhaný a vypíše výsledok binárneho porovnania týchto dvoch súborov.

Testy sa dajú spustiť z koreňového adresáru projektu príkazom:

```
pytest --tb=short
```

Zadanie parametru `--tb=short` docieli, že na štandartný výstup budú vypísané len binárne porovnania pri zlyhaných testoch. Každý testovací scenár má pozitívnu a negatívnu variantu. Pozitívna varianta znamená, že test očakáva, že definované vlastnosti budú v log súbore splnené. Negatívna varianta znamená, že test očakáva, že nejaká z vlastností bude v log súbore nesplnená. Nasleduje tabuľka testovacích scenárov s krátkym popisom.

Tabuľka 8.1: Zoznam testovacích scenárov

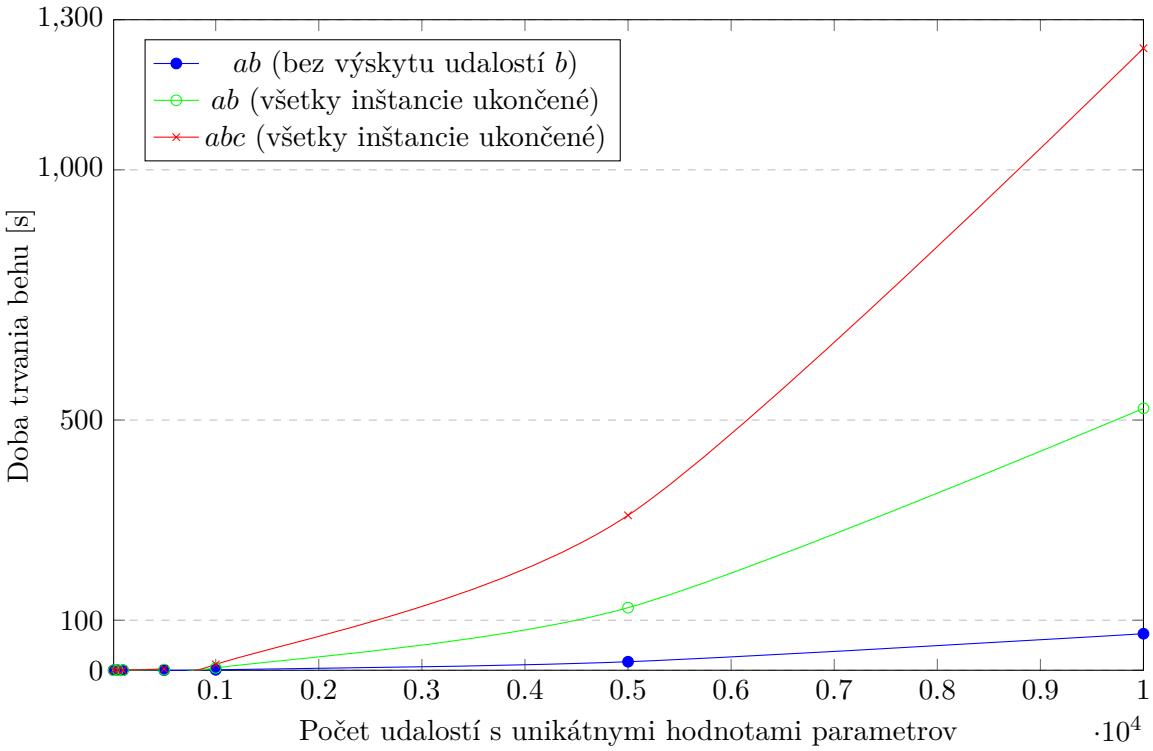
Identifikátor testu	Popis
test01_nonparametric_basic_positive	Log súbor je časť zo *syslog* súboru. Pozitívny test pre overenie postupnosti dvoch udalostí bez parametrických vlastností
test02_nonparametric_basic_negative	Log súbor je časť *syslog* súboru. Negatívny test pre overenie postupnosti dvoch udalostí bez parametrických vlastností
test03_nonparametric_complex_negative	Negatívny test. Log súbor je časť z Windows firewall logu. Overuje jednu pozitívnu a jednu negatívnu vlastnosť. Definícia vlastností obsahuje iteráciu.
test04_nonparametric_complex_positive	Pozitívny test. Log súbor je časť z Windows firewall logu. Overuje jednu pozitívnu a jednu negatívnu vlastnosť. Definícia vlastností obsahuje iteráciu.
test05_parametric_basic_positive	Pozitívny test. Vlastnosť je sekvencia troch udalostí. Overenie parametrických obmedzení.
test06_parametric_basic_negative	Negatívny test. Vlastnosť je sekvencia troch udalostí. Overenie parametrických obmedzení.
test07_parametric_instantiation_negative	Negatívny test. Overenie vytvorenia inštancie pri výskyte novej hodnoty parametra. Overenie obmedzení medzi parametrami.
test08_parametric_instantiation_positive	Pozitívny test. Overenie vytvorenia inštancie pri výskyte novej hodnoty parametra. Overenie obmedzení medzi parametrami.
test09_instantiation_closed_instance_negative	Negatívny test. Overenie vytvorenia novej inštancie pri prázdnom zozname inštancií.
test10_instantiation_closed_instance_positive	Pozitívny test. Overenie vytvorenia novej inštancie pri prázdnom zozname inštancií.
test11_garbage_collector_basic	Overenie sekvenčného uvoľňovania inštancií. Nastavenie parametru *timeout* na 3.
test12_multiple_parameters_in_constraints_p...	Pozitívny test. Overenie, že nástroj spracuje v definícií obmedzení operáciu rovnosti s viac parametrami.
test13_multiple_parameters_in_constraints_n...	Negatívny test. Overenie, že nástroj spracuje v definícií obmedzení operáciu rovnosti s viac parametrami.

Identifikátor testu	Popis
test14_additional_constraint_parameter_pos...	Pozitívny test. Overenie, že ak je v definícií obmedzenia udalosť, ktorá nenastane, tak budú ostané operandy vyhodnotené správne.
test15_additional_constraint_parameter_neg...	Negatívny test. Overenie, že ak je v definícií obmedzenia udalosť, ktorá nenastane, tak budú ostané operandy vyhodnotené správne.
test16_bounded_iteration_positive	Pozitívny test. Test na operáciu obmedzenej iterácie v definície vlastnosti.
test17_bounded_iteration_negative	Negatívny test. Test na operáciu obmedzenej iterácie v definície vlastnosti.
test18_iteration_negative	Negatívny test. Test na operáciu iterácie v definície vlastnosti.
test19_iteration_positive	Pozitívny test. Test na operáciu iterácie v definície vlastnosti.
test20_parametric_nonparametric_simultaneo...	Pozitívny test. Test na overenie, že sa nástroj správne vysporiada s kombináciou parametrických a neparametrických udalostí.
test21_parametric_nonparametric_simultaneo...	Negatívny test. Test na overenie, že sa nástroj správne vysporiada s kombináciou parametrických a neparametrických udalostí.
test22_stdout_log_output_json	Test na funkčnosť prúdového spracovania. Nástroj by mal porušenú vlastnosť hlásiť na štandartný výstup vo formáte JSON.
test23_stdout_log_output_text	Test na funkčnosť prúdového spracovania. Nástroj by mal porušenú vlastnosť hlásiť na štandartný výstup ako text.
test24_garbage_collector_parallel	Overenie funkčnosti pararelného uvoľňovania inštancií konečných automatov.
test25_greater_operator_negative	Negatívny test. Test na funkčnost operátora '>' v definícií obmedzení.
test26_greater_operator_positive	Pozitívny test. Test na funkčnost operátora '>' v definícií obmedzení.
test27_greater_or_equal_operator_negative	Negatívny test. Test na funkčnost operátora '>=' v definícií obmedzení.
test28_greater_or_equal_operator_positive	Pozitívny test. Test na funkčnost operátora '>=' v definícií obmedzení.
test29_less_operator_negative	Negatívny test. Test na funkčnost operátora '<' v definícií obmedzení.
test30_less_operator_positive	Pozitívny test. Test na funkčnost operátora '<' v definícií obmedzení.
test31_less_or_equal_operator_negative	Negatívny test. Test na funkčnost operátora '<=' v definícií obmedzení.
test32_less_or_equal_operator_positive	Negatívny test. Test na funkčnost operátora '<=' v definícií obmedzení.

Identifikátor testu	Popis
test33_multiple_duplications_negative	Negatívny test. Overenie duplikovania inštancie z konfigurácie vzniknutej duplikovaním
test34_multiple_duplications_positive	Pozitívny test. Overenie duplikovania inštancie z konfigurácie vzniknutej duplikovaním
test35_multiple_overlapping_properties_nega...	Negatívny test. Overenie, že keď sa udalosti vyskytnú v niekoľkých vlastnostiach, tak sa nástroj zachová správne.
test36_multiple_overlapping_properties_posi...	Pozitívny test. Overenie, že keď sa udalosti vyskytnú v niekoľkých vlastnostiach, tak sa nástroj zachová správne.

8.2 Testovanie výkonnosti

Testovanie výkonnosti bolo realizované na troch scenároch s rôznymi počtami udalostí. Prvý scenár obsahoval jednu definovanú vlastnosť ab a jedno obmedzenie $a.1 = b.1$. Udalosti boli definované v tvare $a|b\ num$, kde num bolo celé číslo. V prvom testovacom scenári sa nevyskytovala žiadna udalosť b. Sled udalostí obsahoval len udalosť a s unikátnou hodnotou celočíselného parametru. Po dokončení zostal teda nástroj s otvorenými všetkými inštanciami konečných automatov. Nástroj bol spustený s 10, 50, 100, 500, 1000, 5000 a 10000 udalosťami.

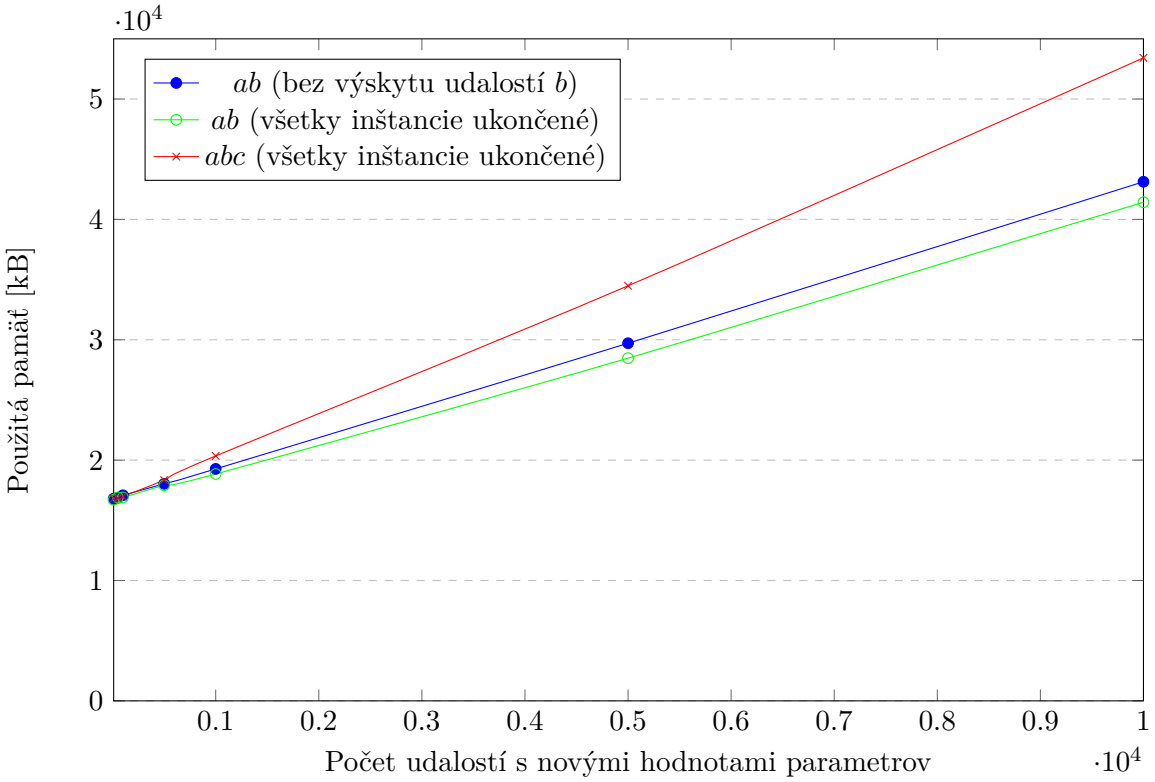


Obr. 8.1: Doba behu nástroja v závislosti od počtu udalostí s unikátnymi hodnotami

Druhý scenár obsahoval rovnakú definíciu vlastností a obmedzení. Rozdiel oproti prvému scenáru bol ten, že po n udalostiach a s unikátnymi hodnotami parametrov nastalo n udalostí b s odpovedajúcimi hodnotami. Všetky inštancie boli teda ukončené a definovaná vlastnosť splnená.

Tretí scenár pozostával z jednej definovanej vlastnosti abc a obmedzenia $a.1 = b.1 = c.1$. Sekvencia udalostí obsahovala n udalostí a s unikátnymi parametrami, následne sa vyskytlo n udalostí b s odpovedajúcimi hodnotami parametrov a nakoniec nastalo n udalostí c s hodnotami, ktoré vyhovovali obmedzeniu.

Na obrázku 8.1 je graf znázorňujúci dobu behu pre tri spomínané scenáre. Modrá krivka odpovedá prvému, zelená druhému a červená tretiemu scenáru. Tretí scenár sa logicky ukázal ako najviac časovo náročný. V tomto scenári sa pri každej vyskytnutej udalosti c kontrolovali nielen všetky otvorené inštancie, ale aj minulé konfigurácie všetkých inštancí. Aj ked išlo len o maximálne dve konfigurácie pre každú inštanciu a teda konfiguračné stromy neboli skoro vôbec rozvetvené (listové uzly monitor nekontroluje), dobu behu to značne predĺžilo.



Obr. 8.2: Množstvo použitej pamäte v závislosti od počtu udalostí s unikátnymi hodnotami

Na obrázku 8.2 je znázornené množstvo použitej pamäte v závislosti od počtu udalostí s unikátnymi hodnotami. Scenáre použité v tomto grafe sú rovnaké ako v predchádzajúcom. Tretí scenár sa ukázal ako najviac pamäťovo náročný. Veľký rozdiel v množstve použitej pamäte u tohto scenára v porovnaní s ostatnými dvoma scenármi je spôsobený vyšším počtom uložených konfigurácií inštancií konečných automatov. Počet inštancií je v porovnaní s ostatnými dvoma scenármi rovnaký. Líši sa len počet validných prechodov a pre každý validný prechod je vytvorený nový konfiguračný uzol.

Hodnoty boli získané príkazom `usr/bin/time`. Množstvo použitej pamäte z položky *maximum resident set size* a doba behu nástroja z položky *user time*. Všetky hodnoty boli namerané na procesore *Intel Core i5 3210M @ 2.50GHz*.

Kapitola 9

Záver

V tejto práci boli spomenuté základné informácie o verifikácií programu za behu. Boli popísané dva spôsoby špecifikácie sekvencií udalostí a to regulárne výrazy a výrazy temporálnej logiky vrátane ich zaradenia do Chomského hierarchie. Bola priblížená problematika parametrických vlastností udalostí. Tiež boli popísané štyri existujúce nástroje riešiace verifikáciu záznamov s parametrickými vlastnosťami.

Výsledný nástroj *plogchecker* je postavený na existujúcom nástroji *logchecker* [12]. Podporuje definovanie obmedzení nad parametrami udalostí a overenie týchto obmedzení. Pre špecifikáciu sekvencií udalostí využíva rozšírené regulárne výrazy. Tieto regulárne výrazy sú transformované na deterministické konečné automaty, ktoré sú schopné rozhodnúť, či daná sekvencia určitú vlastnosť spĺňa, alebo ju porušuje. Nástroj dokáže verifikovať parametrické vlastnosti nad sekvenciami udalostí vyskytnutých v log súbore. Užívateľ môže definovať závislosti medzi parametrami udalostí. Výstupom nástroja je report o porušených vlastnostiach so znázornými sekvenciami udalostí, ktoré porušenie vlastnosti spôsobili.

Nevýhodou výsledného nástroja je jeho vysoká časová a pamäťová náročnosť. Riešením pre vysokú časovú náročnosť by bolo použitie paralelizmu, prípadne implementácia nástroja v kompilovanom jazyku. Ako eventuálne rozšírenie by bolo vhodné implementovať podporu pre spôsob špecifikácie s vyššou vyjadrovaciou schopnosťou ako regulárne výrazy. Pre vyššiu užívateľskú prívetivosť by bola vhodná podpora pre komplexnejšie výrazy v definícií obmedzení.

Literatúra

[1] *Regular expression operations* [online]. [cit. 2020-05-21]. Dostupné z: https://docs.python.org/3/library/re.html.

[2] *Regular Expressions* [online]. [cit. 2020-05-18]. Dostupné z: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html.

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

[4] Barringer, H., Havelund, K., Rydeheard, D. a Groce, A. *Rule Systems for Runtime Verification: A Short Tutorial*. Springer, Berlin, Heidelberg, 2009. ISBN 978-3-642-04693-3.

[5] Bartocci, E., Falcone, Y., Francalanza, A. a Reger, G. Introduction to Runtime Verification. In:. Február 2018, s. 1–33. ISBN 978-3-319-75631-8.

[6] Beazley, D. M. *PLY (Python Lex-Yacc)* [online]. [cit. 2020-01-15]. Dostupné z: http://www.dabeaz.com/ply/ply.html.

[7] Caleb, E. *Automata* [online]. 2019 [cit. 2019-10-02]. Dostupné z: https://github.com/caleb531/automata.

[8] Diekert, V. a Gastin, P. First-order definable languages. In:. Január 2008, s. 261–306.

[9] Falcone, Y., Havelund, K. a Reger, G. A Tutorial on Runtime Verification. IOS Press. 2013, roč. 34, s. 141–175. Summer School Marktoberdorf 2012. Dostupné z: https://hal.inria.fr/hal-00853727.

[10] Havelund, K. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*. 2015, roč. 17, č. 2, s. 143–170. Dostupné z: https://app.dimensions.ai/details/publication/pub.1026729420.

[11] Johnson, S. C. *Yacc Meets C++* [online]. [cit. 2020-01-15]. Dostupné z: http://yaccman.com/papers/YPP.html.

[12] Kapičák, P. *Nástroj pro ladění post-mortem*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/121836.

[13] Krekel, H. *Pytest: helps you write better programs - pytest documentation* [online]. [cit. 2020-05-19]. Dostupné z: https://docs.pytest.org.

[14] Mukund, M. *Linear-time temporal logic and Büchi automata.* 1997.

[15] Mukund, M. Finite-State Automata on Infinite Inputs. August 2001.

[16] Sipser, M. *Introduction to the Theory of Computation.* 1st. International Thomson Publishing, 1996. ISBN 053494728X.

Príloha A

Finálny report špecifikovaný v JSON Schema

```
{
 "type": "object",
 "properties": {
  "properties": {
   "type": "object",
   "properties": {
    "id": {
     "type": "object",
     "properties": {
      "property": {
       "type": "string"
      },
      "violated": {
       "type": "array",
       "items": {
        "type": "object",
        "properties": {
         "id": {
          "type": "string"
         },
         "is_property_met": {
          "type": "boolean"
         },
         "events_sequence": {
          "type": "array",
          "items": {
           "type": "object",
           "properties": {
            "event_id": {
             "type": "string"
            },
            "log_file": {
             "type": "string"
            },
            "log_lineno": {
             "type": "integer"
```

```
                    },
                    "log_line": {
                      "type": "string"
                    }
                  }
                }
              }
            }
          }
        }
      }
    },
    "badproperties": {
      "type": "object",
      "properties": {
        "id": {
          "type": "object",
          "properties": {
            "property": {
              "type": "string"
            },
            "violated": {
              "type": "array",
              "items": {
                "type": "object",
                "properties": {
                  "id": {
                    "type": "string"
                  },
                  "is_property_met": {
                    "type": "boolean"
                  },
                  "events_sequence": {
                    "type": "array",
                    "items": {
                      "type": "object",
                      "properties": {
                        "event_id": {
                          "type": "string"
                        },
                        "log_file": {
                          "type": "string"
                        },
                        "log_lineno": {
                          "type": "integer"
                        },
                        "log_line": {
                          "type": "string"
                        }
                      }
                    }
                  }
```

```
91                  }
92                }
93              }
94            }
95          }
96        }
97      }
98    }
99  }
```

Výpis A.1: Forma finálneho reportu vyjadrená v JSON Schema

Príloha B

Overenie výstupov z projektu pre predmet IOS nástrojom plogchecker

B.1 Zadanie druhého projektu z roku 2015/2016

Existujú dva typy procesov, vozík a pasažieri. Pasažieri nastupujú do vozíku, ktorý má obmedzenú kapacitu. Akonáhle je vozík plný, vyráža na trať. Ďalší pasažieri musia počkať, kým sa vozík vráti a všetci z neho vystúpia.

B.1.1 Spustenie

```
./proj2 P C PT RT
```

kde

- `P` je počet procesov reprezentujúcich pasažiera, `P > 0`
- `C` je kapacita vozíku; `C > 0, P > C`, `P` musí byť vždy násobkom `C`.
- `PT` je maximálna hodnota doby (v milisekundách), po ktorej je generovaný nový proces pre pasažiera; `PT >= 0 && PT < 5001`
- Všetky parametre sú celé čísla

B.1.2 Popis procesov a ich výstupov

Poznámky k výstupom:

- `A` je poradové číslo vykonávanej akcie,
- `NAME` je skratka kategórie príslušného procesu, tj. `P` alebo `C`,
- `I` je interný identifikátor procesu identifikátor procesu v rámci príslušnej kategórie.

Proces `car`

1. Po spustení vypisuje `A: NAME I: started`

2. Proces pracuje v cykle, počet iterácií je `P/C`
3. V každej iterácií:
   (a) proces vyvolá operáciu *load* (zahájenie nastupovanie); vypisuje `A: NAME I: load`
   (b) proces čaká, kým nenastúpi všetci pasažieri
   (c) ak je vozík plný
      i. ďalší pasažieri nemôžu nastúpiť
      ii. proces vyvolá operáciu *run* (jazda vozíku); vypisuje `A: NAME I: run`
      iii. proces sa uspí na náhodnú dobu z intervalu `<0, RT>`
   (d) do prebudenia vyvolá proces operáciu *unload* (zahájenie vystupovania); vypisuje `A: NAME I: unload`
   (e) proces nemôže zahájiť operáciu *load*, kým nevystúpia všetci pasažieri
4. Tesne pred ukončením proces vypisuje `A: NAME I: finished`.

Proces *passenger*

1. Po spustení vypisuje `A: NAME I: started`
2. Proces čaká na splnenie podmienok nastupovania, proces vyvolá operáciu *board* (nastúpenie do vozíku)
   (a) vypisuje `A: NAME I: board`
   (b) ak nie je proces posledný nastupujúci pasažier, vypisuje `A: NAME I: board order N`, kde `N` je poradie, v ktorom proces nastúpil do vozíku
   (c) ak je proces posledný nastupujúci pasažier, vypisuje `A: NAME I: board last`
3. Potom, čo proces *car* vyvolá operáciu unload
   (a) proces zaháj operáciu *unboard* (vystúpenie z vozíku); `A: NAME I: unboad`
   (b) ak nie je proces posledný vystupujúci pasažier, vypisuje `A: NAME I: unboard order N`, kde `N` je poradie, v ktorom proces vystúpil z vozíku
   (c) ak je proces posledný vystupujúci pasažier, vypisuje `A: NAME I: unboard last`
4. Tesne pred ukončením proces vypisuje `A: NAME I: finished`.

Spoločné podmienky

1. Všetky procesy *passenger* a *car* sa ukončia súčasne, tj. čakajú, kým všetci dokončia operáciu *unboard*, alebo hlavný cyklus. Až potom vypíšu informáciu `... finished`.
2. Ak má `RT` alebo `PT` hodnotu 0, znamená to, že na prislušnom mieste nedojde k uspaniu procesu.

B.2 Definícia vlastností v nástroji *plogchecker*

```
properties:
  # sekvencia car procesu (v zlozenych zatvorkach by mala byt hodnota P/C)
  p1: a(bcd){2}e
  # sekvencia passenger procesu
  p2: fg(h|i)j(k|l)m
badproperties:
  # overenie, ze nie je vypis opakovany
  p3: (aa)|(ee)|(ff)|(gg)|(hh)|(ii)|(jj)|(kk)|(ll)|(mm)
events:
  a: '\\d+ \: C (\\d+) \\: started$'
  b: '\\d+ \: C (\d+) \: load$'
  c: '\\d+ \: C (\d+) \: run$'
  d: '\\d+ \: C (\d+) \: unload$'
  e: '\\d+ \: C (\d+) \: finished$'
  f: '\\d+ \: P (\d+) \: started$'
  g: '\\d+ \: P (\d+) \: board$'
  h: '\\d+ \: P (\d+) \: board order \d+$'
  i: '\\d+ \: P (\d+) \: board last$'
  j: '\\d+ \: P (\d+) \: unboard$'
  k: '\\d+ \: P (\d+) \: unboard order \d+$'
  l: '\\d+ \: P (\d+) \: unboard last$'
  m: '\\d+ \: P (\d+) \: finished$'

constraints:
- "a.1 = b.1 = c.1 = d.1 = e.1"
- "f.1 = g.1 = h.1 = i.1 = j.1 = k.1 = l.1 = m.1"
```

B.3 Príklad správnej sekvencie udalostí

```
1 : C 1 : started
2 : C 1 : load
3 : P 1 : started
4 : P 1 : board
5 : P 1 : board order 1
6 : P 2 : started
7 : P 2 : board
8 : P 2 : board last
9 : C 1 : run
10 : P 3 : started
11 : C 1 : unload
12 : P 1 : unboard
13 : P 1 : unboard order 1
14 : P 4 : started
15 : P 2 : unboard
16 : P 2 : unboard last
```

```
17 : C 1 : load
18 : P 4 : board
19 : P 3 : board
20 : P 4 : board order 1
21 : P 3 : board last
22 : C 1 : run
23 : C 1 : unload
24 : P 4 : unboard
25 : P 4 : unboard order 1
26 : P 3 : unboard
27 : P 3 : unboard last
28 : P 4 : finished
29 : C 1 : finished
30 : P 1 : finished
31 : P 3 : finished
32 : P 2 : finished
```

B.3.1 Výsledný report vo formáte JSON

```
{
   "properties": {
      "p1": {
         "property": "a(bcd){2}e",
         "violated": []
      },
      "p2": {
         "property": "fg(h|i)j(k|l)m",
         "violated": []
      }
   },
   "badproperties": {
      "p3": {
         "property": "(aa)|(ee)|(ff)|(gg)|(hh)|(ii)|(jj)|(kk)|(ll)|(mm)",
         "violated": []
      }
   }
}
```

B.4 Príklad nesprávnej sekvencie udalostí

```
1 : C 1 : started
2 : C 1 : load
3 : P 1 : started
4 : P 1 : board
5 : P 1 : board order 1
6 : P 2 : started
7 : P 2 : board
8 : P 2 : board last
```

```
9 : C 1 : run
10 : P 3 : started
11 : C 1 : unload
12 : P 1 : unboard order 1
13 : P 4 : started
14 : P 2 : unboard
15 : P 2 : unboard last
16 : C 1 : load
17 : P 4 : board
18 : P 3 : board
19 : P 4 : board order 1
20 : P 3 : board last
21 : C 1 : run
22 : C 1 : unload
23 : P 4 : unboard
24 : P 4 : unboard order 1
25 : P 3 : unboard
26 : P 4 : unboard
27 : P 3 : unboard last
28 : P 4 : finished
29 : C 1 : finished
30 : P 1 : finished
31 : P 3 : finished
32 : P 2 : finished
```

B.4.1 Výsledný report vo formáte JSON

```
1  {
2     "properties": {
3       "p1": {
4          "property": "a(bcd){2}e",
5          "violated": []
6       },
7       "p2": {
8          "property": "fg(h|i)j(k|l)m",
9          "violated": [
10            {
11               "id": "automaton_3",
12               "is_property_met": false,
13               "events_sequence": [
14                  {
15                     "event_id": "f",
16                     "log_file": "tests/logs/IOS",
17                     "log_lineno": 3,
18                     "log_line": "3 :P 1:started"
19                  },
20                  {
21                     "event_id": "g",
22                     "log_file": "tests/logs/IOS",
23                     "log_lineno": 4,
```

```
                  "log_line": "4 :P 1:board"
               },
               {
                  "event_id": "h",
                  "log_file": "tests/logs/IOS",
                  "log_lineno": 5,
                  "log_line": "5 :P 1:board order 1"
               }
            ]
         }
      ]
   }
},
"badproperties": {
   "p3": {
      "property": "(aa)|(ee)|(ff)|(gg)|(hh)|(ii)|(jj)|(kk)|(ll)|(mm)",
      "violated": [
         {
            "id": "automaton_22",
            "is_property_met": true,
            "events_sequence": [
               {
                  "event_id": "j",
                  "log_file": "tests/logs/IOS",
                  "log_lineno": 23,
                  "log_line": "23 :P 4:unboard"
               },
               {
                  "event_id": "j",
                  "log_file": "tests/logs/IOS",
                  "log_lineno": 26,
                  "log_line": "26 :P 4:unboard"
               }
            ]
         }
      ]
   }
}
```