VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRO LADĚNÍ POST-MORTEM

A TOOL FOR POST-MORTEM DEBIGGING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **PETER KAPIČÁK**
AUTHOR

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

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Kapičák Peter**
Program: Informační technologie
Název: **Nástroj pro ladění post-mortem**
A Tool for Post-Mortem Debugging
Kategorie: Analýza a testování softwaru

Zadání:

1. Nastudujte techniky ladění programů. Seznamte se s výpisy určené pro post-mortem ladění programů, tj. ladění programů po ukončení jejich běhu pomocí ladicích výpisů.
2. Analyzujte požadavky na analýzu konečné stopy programů. Navrhněte programové vybavení pro analýzu stopy programů. Program by měl umožnit uživateli specifikovat ověřované vlastnosti nad stopou programu.
3. Implementujte nástroj pro post-mortem analýzu konečné stopy programů. Nástroj integrujte pro platformu Testos.
4. Ověřte funkcionalitu nástroje na umělé sadě různých stop programů a různých ověřovaných vlastností.

Literatura:

- Domovská stránka projektu Kint, url: http://raveren.github.io/kint/
- Manuálová stránka debug_backtract: url: http://php.net/manual/en/function.debug-backtrace.php
- Modul trace pro Python 2.x, url: https://docs.python.org/2/library/trace.html

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Cieľom tejto práce je kontrolovať užívateľom špecifikované vlastnosti nad stopou programu alebo nad log súborom, ktoré by mal program spĺňať alebo naopak, ktoré by nemal spĺňať. Vlastnosti a ich opis sú základom nástroja pre ladenie post-mortem. Sú transformované na deterministický konečný automat aby sa dali overovať v stopách programov a ich opis je dôležitý pre vyhľadávanie konkrétnych udalostí v stopách programov, ktoré sú automatu posielané na vstup. Výstupom nástroja je výsledná správa o tom či boli vlastnosti splnené alebo porušené. Vytvorené riešenie poskytuje overovanie vlastností stôp programov, log súborov bez ohľadu na ich formát a aké udalosti predchádzali porušeniu vlastnosti.

Abstract

The goal of this work is checking specific properties in trace which program should meet or which it shouldn't meet. Properties and their description are basis of this tool for post-mortem debbuging. Properties are transformed to deterministic finite automaton for checking in trace and their description is important for searching events in trace. Events are pass to automaton as input. Output of a tool is report if properties are meet or not. Solution provides check of properties regardless of log format and shows events that preceded violation of the properties.

Kľúčové slová

vlastnosti, log, analýza, ladenie, post-mortem, Testos, Python

Keywords

properties, log, analysis, debugging, post-mortem, Testos, Python

Citácia

KAPIČÁK, Peter. *Nástroj pro ladění post-mortem.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Nástroj pro ladění post-mortem

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pana Ing. Aleša Smrčku, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Peter Kapičák
15. mája 2019

Poďakovanie

Chcel by som sa poďakovať môjmu vedúcemu práce pánovi Ing. Alešovi Smrčkovi, Ph.D. za poskytnutie literatúry, odborné konzultácie, cenné rady, ochotu a vedenie pri tvorbe tejto práce.

Obsah

Kapitola 1

Úvod

S postupným vývojom stále nových technológií, softvérov sa kladú čoraz väčšie nároky na testovanie. To je ale často finančne a časovo náročné. Ladenie post-mortem, ktorým sa zaoberá táto práca má výhody, že chyby nemusia byť skúmané počas činnosti softvéru, ale až po jeho skončení. Rôzne informácie zozbierané pri činnosti softvéru môžu byť skúmané a môžu byť tak odhalené chyby, ktoré sa pri činnosti softvéru ani neprejavili.

Tento typ testovania ešte nie je plne automatizovaný pretože to je náročná úloha. Správy vytvorené počas behu softvéru o tom aké udalosti sa v ňom diali nemajú štandardizovaný formát a preto je automatizovanie náročné. Vývojár pri tomto ladení musí prechádzať správy manuálne. Ladenie má však v určitých prípadoch nespochybniteľné výhody nakoľko iný typ ladenia softvéru objaví chybu na určirom mieste, no to nemusí znamenať, že jej neprechádzala iná chyba a práve ladenie post-mortem vie odhaliť tieto chyby pretože má k dispozícií informácie o celom behu softvéru na jednom mieste.

Tento nástroj je jedna z utilít platformy Testos, ktorý je sada nástorojov automatizujúcich testovanie. Nástroj pre ladenie post-mortem som si vybral pretože toto testovanie má už spomenuté výhody a pre zjednodušenie práce vývojárom aby nebolo potrebné odhalovať chyby manuálne ale definovať si vlastnosti, ktoré má softvér spĺňať alebo naopak, ktoré nesmie spĺňať. Nástroj tieto vlastnosti skúma v správach (log súbroch) a pomáha odhaľovať aké udalosti predchádzali chybe alebo prečo sa nejaká akcia nevykonala správne.

V kapitole 2 je analýza oblasti, ktorou sa práca zaoberá a analýza technológií, ktoré práca využíva. Pokračuje návrhom samotného nástroja a analýzou súčasného stavu v kapitole 3. Implementácia a čo k nej bolo použité ako aj testovanie nástroja je v kapitole 4. V poslednej kapitole 5 je zhrnutie celej práce.

Kapitola 2

Analýza existujúcich riešení a použitých technológií

2.1 Log súbory

Log súbory obsahujú informácie a udalosti o jednom špecifickom systéme. Analýza problémov z týchto súborov je preto užitočná, no háčik to má keď problém zahŕňa niekoľko systémov a zariadení.

Neexistuje totižto žiadny štandard pre umiestnenie, použiteľnosť, formát a veľkosť log súborov, ktorý musí byť rovnaký na rozdielnych systémoch a zariadeniach. Tento problém je ďalej riešený v sekciách 2.1 a 2.1.

To limituje aj maximum informácií extrahovaných z log súborov a robí zložitejšiu ich analýzu.

Log súbor je textový súbor používaný na uchovávanie automaticky vygenerovaných udalostí, chovania a podmienok označených časovým razítkom relevantných pre určitý systém [10].
Príklady, kde sa využívajú log súbory a ako:

- **webový server** - zaznamenáva počet prístupov na každú stránku, koľko užívateľov navštívilo stránku a z akej domény atď.
- **operačný systém** - zaznamenáva upozornenia, užívateľksé prístupy, chyby, udalosti, informácie o procesoch.
- **sieťové prvky** (smerovač, prepínač) - zaznamenáva či je port aktívny alebo neaktívny, aká služba je aktívna, ktorá komunikácia bola povolená a ktorá nie atď.
- **stopa programu** - systémové volania, počas vykonávania programu.
- **zásobník volaní** - zásobník volaní užívateľom definovaných funkcií.

Využitie informácií z log súboru je pre rôzne účely. Môže poslúžiť pre ladenie programov, aká chyba sa vyskytla v operačnom systéme, slúži pre odhaľovanie útokov napríklad na webový server. Dajú sa z neho extrahovať štatistiky zo siete a to veľkosť stiahnutých a nahraných dát za určité obdobie, v ktorom čase bola najväčšia vyťaženosť na uzloch, čo môže pomôcť k prehodnoteniu rozšíriť sieť a pod.

Log manažment

Log manažment je oblasť riešiaca tri základné problémy, ktoré pri log súboroch nastávajú:

- generovanie log súborov a ich úložisko - log súbory musí niekto vytvárať a taktiež ich musí zhromažďovať a ukladať,
- ochrana log súborov - log súbory môžu obsahovať citlivé informácie, ktoré nesmú byť prístupné každému,
- analýza - získavanie dát z log súborov, vytváranie štatistík, rôzne prognózy do budúcna atď.

Zbieranie a ukladanie log súborov v sebe zahŕňa riešenia ako efektívne ukladať log súbory pretože ide o veľké objemy dát a navyše sa musia podľa zákonov danej krajiny uchovávať určité obdobie. Jedna zo základných techník je rotácia log súborov.
Analýza a monitorovanie log súborov znamená operácie ako redukcia log súborov pretože nie všetky zaznamenané udalosti sú relevantné, často sú udalosti redundantné, takže ich stačí reprezentovať jednou udalosťou a pod. Zahŕňa to získavanie rôznych štatistík napríklad o veľkosti prenosovej rýchlosti na sieti a rôzne ďalšie štatistiky pre adminstrátora, ktoré môžu odhalovať chyby a slabé miesta. Vytváranie správ o analýze log súborov je ďalšia možnosť, ktorú rieši log manažment.

Ide o rozsiahlu tému, o ktorej sa dá viac dozvedieť tu [6].

Existujúce log formáty

Postupným vývojom log manažmentu a nástrojov, ktoré ho implementujú dnes už pokročilými technikami narástla oblasť okolo ukladania udalostí z rôznych programov, služieb, sietí do solídnych rozmerov. Jeden problém ale cez to všetko pretrváva. Log formát nemá pevne definovaný štandard a už bolo vytvorených niekoľko štandardov, čo naznačuje aspoň dobrý smer k jednému štandardu.

Pre príklad sú uvedené tri štandardy podľa článku [2]:

- CEE (Common event expression) od firmy Mitre - v súčasnosti sa už nepracuje na jeho vývoji pretože projekt prestal byť financovaný,
- OLF (Open log format) má niekoľko nevýhod, napr. nie je jasné pre čo je štandard určený, pri každej udalosti treba zaznamenať IP adresy avšak nie vždy ich treba, udalosti o chybe nie sú odlišné od klasických a ďalšie nevýhody sú uvedené tu [8],
- CEF (Common event format) je dodnes používaný štandard predovšetkým pre udalosti týkajúce sa bezpečnosti [1], v minulosti vytvorený firmou ArcSight.

2.2 Post-mortem ladenie

Ladenie post-mortem je jedna z ladiacich techník pre zisťovanie chýb v zdrojovom kóde nejakého programovacieho jazyku. Zatiaľ čo ostatné ladiace techniky sú vykonávané počas behu programu a dajú sa nájsť chyby iba počaš jeho behu, tak ladenie post-mortem je

technika zisťovania chýb v programe až po jeho skončení.

Použitie

Ladnie post-mortem znamená analýzu rôznych záznamov, ktoré môžu byť vytvorené počas behu programu a ostanú uložené ako súbory aj po skončení programu. Ide o zázanamy ako zásobník volaní, pamäťová stopa programu a iné. Nie všetky programovacie jazyky alebo prostredia zhromažďujú počas behu programu tieto záznamy automaticky takže je problematické sa k nim dostať, ale dnes už existuje množstvo rozšírení pre programovacie jazyky alebo samostatné nástroje zhromažďujúce tieto záznamy. Niektoré vývojové prostredia vedia počas behu programu zastaviť beh pri nejakej chybe a presunúť sa do ladiaceho režimu, kde má užívateľ k dispozícií tieto záznamy a vie sa dopátrať k chybe a čo ju spôsobilo prípadne [7].

Výhody tejto ladiacej techniky sú v rýchlom zistení, kde nastala chyba a vie určiť miesto v zdrojovom kóde. Avšak chyba na určenom mieste nemusí znamenať, že jej nepredchádzala iná chyba, čo už môže byť problém dohľadať.

Nevýhodou naopak môže byť tažká interpretácia záznamov, nie je možné odhaliť napríklad logickú chybu alebo chybu v UX, nedostatok záznamov, toto ladenie vyžaduje určitú úroveň poznatkov v danej oblasti alebo v zdrojových súboroch [4].

2.3 Lexikálna a syntaktická analýza

Jednoduché lexikálne a syntaktické analýzy je možné si implementovať. Problém nastáva ak treba niečo pozmeniť. Vtedy sú nevyhnutné zásahy do veľkej časti implementácie a v neposlednom rade tým programátor stráca čas, ktorý mohol využiť na ďalšie veci. Aj kvôli tomuto problému existujú nástroje lex a yacc nahrádzajúce vlastné riešenia lexikálnej a syntaktickej analýzy.

Pri nasledujúcom opise oboch nástrojov som vychádzal z knihy [5], kde sa k obom nástrojom môžno dočítať viac.

Lex

Lex je lexikálna analýza, ktorá delí vstup na časti, inak nazvané lexémy (angl. tokens). Každý lexém má svoju definíciu vo forme regulárneho výrazu, ktorá sa nazýva lex špecifikácia. Lex vie tieto regulárne výrazy previesť do formy aby vyhľadával vo vstupnej sekvencií veľmi rýchlo nezávisle na počte výrazov, ktoré sa snaží nájsť.

Yacc

Tento nástroj slúži ako syntaktická analýza. Spája lexémy z lexikálnej analýzy, pretože má definované vzťahy medzi nimi prostredníctvom pravidiel. Yacc prijíma ako vstup sekvenciu lexémov a zisťuje či sa zhodujú s nejakým pravidlom z gramatiky. Je možné, že bude pomalší ako vlastnoručne implementovaná syntaktická analýza ale má výhodu pri úprave pravidiel. Pretože stačí zmeniť gramatiku a yacc sa s tým vysporiada sám, kde naopak pri

vlastnej implementácií by to s každou zmenou vyžadovali častokrát nie malé zásahy do implementácie.

2.4 Existujúce riešenia

V tejto časti sú popísané Existujúce riešenia pre log manažment, vytváranie grafov, štatistík a aj nástroje, kde je používané ladenie post-mortem.

Nástroje pre log manažment

Na dnešnom trhu existuje množstvo nástrojov, ktoré riešia poblematiku log manažmentu, zbierania štatistík z log súborov, upozorňovanie na možné riziká.

Jeden z popredných nástrojov na dnešnom trhu je nepochybne Datadog[1], ktorý okrem log manažmentu ponúka obrovský výber služieb. Samotný log manažment ponúka na jednom mieste zhromažďovanie log súborov, ich spracovanie, v reálnom čase overovať či bol napríklad štart servera vykonaný správne, archiváciu log súborov a možné nastavenie toho ako sa budú log súbory spracovávať a v akom poradí sa čo nad nimi bude vykonávať.

Druhý nástroj na pre log manažment je Graylog[2]. Má obdobnú funkcionalitu a možnosti ako napríklad Datadog alebo iné nástroje na trhu, ale má výhodu, že je voľne dostupný.

Možnosti pre ladenie post-mortem

Vývojové prostredia pre implementáciu obrovského množstva rôznych programov atď. sú dnes veľmi rozšírené. Nie každé vývojové prostredie má kavlitné prostredie pre ladenie, no jedno z tých, ktoré má kvalitnú podporu pre ladenie je Visual Studio od spoločnosti Microsoft. Uvádzam ho ako príklad pretože má grafické rozhranie a umožňuje ladenie post-mortem jednoduchým spôsobom. Počas behu programu zastaví jeho beh a na mieste, kde k chybe došlo poskytne aj rôzne záznamy ako napr. ako zásobník volaní.

[1] https://docs.datadoghq.com/logs/?tab=ussite
[2] https://www.graylog.org/products/open-source

Kapitola 3

Návrh nástroja pre ladenie post-mortem

V tejto kapitole je popísaný návrh nástroja pre ladenie post-mortem, kde sú znázornené dátové toky medzi jednotlivými časťami nástroja. Kapitola začína celkovým návrhom nástroja, ktorého jednotlivé časti sú následne vysvetlené osobitne.

3.1 Zhodnotenie súčasného stavu

Článok o stope (napr. stopa programu) [9] píše o probléme rôznych formátov stôp z pohľadu verifikácie v reálnom čase. Každá služba alebo aplikácia má svoj vlastný formát udalosti v stope, pretože neexistuje žiadny štandard, ktorý by definoval čo má presne obsahovať. Stopy môžu obsahovať dátum, čas a dáta, čo sú najzákladnejšie informácie čo by mala udalosť obsahovať a ďalšie informácie ako premenlivý počet parametrov, jednoduché a zložité dátové štruktúry ostávajú predmetom diskusie.

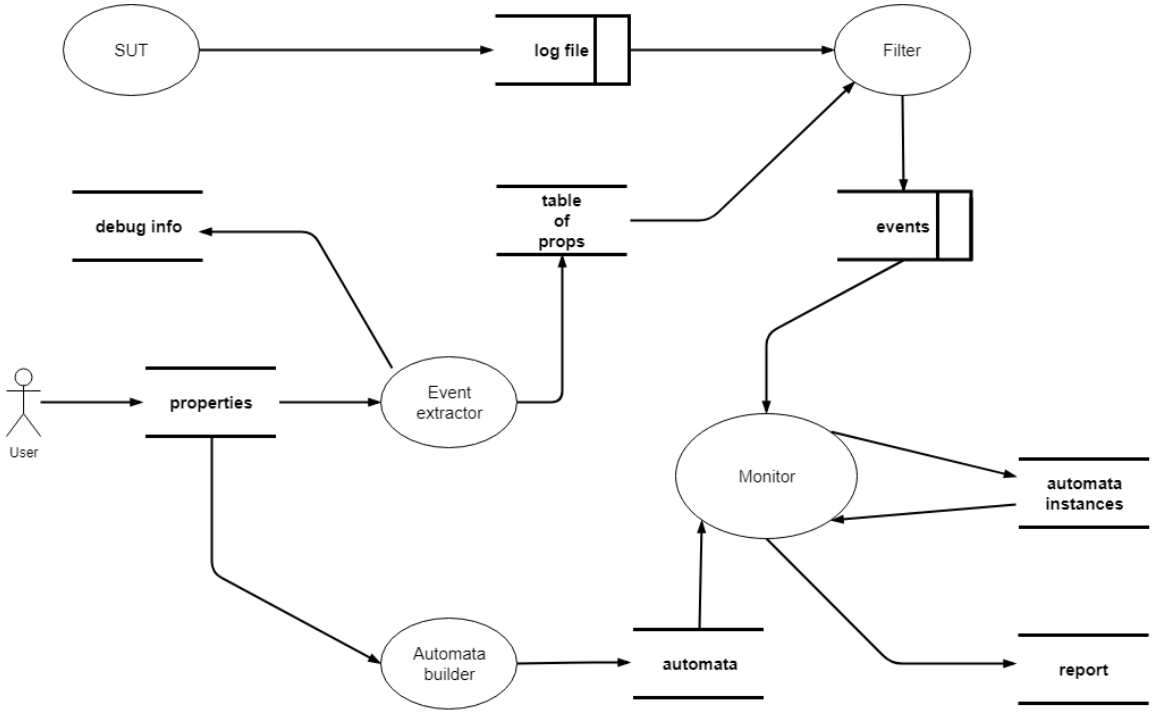
Na základe článku konštatujem, že aj dnes je stále predmetom diskusie, čo by mal byť štandardný formát stôp. Mal by byť dobre definovaný napríklad kvôli syntaktickej analýze. Ďalej mi potvrdzuje návrh mojej práce, že je potrebné definovať rôzne typy udalostí a teda stôp. Návrh je opísaný v nasledujúcich sekciách.

3.2 Špecifikácia požiadaviek

Požiadavky v tabuľke 3.1 popisujú základné nároky na nástroj. Sú členené do kategórií na funkcionálne, požiadavky na projekt a požiadavky na použiteľnosť. Požiadavky majú hierarchickú štruktúru a môžu jeden na druhom závisieť.

3.3 Celkový návrh nástroja

Nástroj prijíma na vstup súbor s popisom vlastností 3.5, ktoré má kontrolovať a taktiež log súbor alebo viacero log súborov, v ktorých má byť vykonaná kontrola vlastností. Pre každú vlastnosť je vytvorený deterministický konečný automat 3.4. Ďalšou častou nástroja je filter log súborov. Ten na základe tabuľky vlastností vyhľadáva udalosti. Vlastnosť je



Obr. 3.1: Diagram dátových tokov - Súčinnosť jednotlivých častí nástroja

sekvencia udalostí a tak jedna udalosť nájdená v log súbore môže spustiť kontrolu jednej zo zadaných vlastností alebo posunúť kontrolu už kontrolovanej vlastnosti o jednu udalosť dopredu. Inak povedané udalosť nájdená v log súbore môže spustiť nový automat alebo prejsť do ďalšieho stavu v už spustenom automate.

Takáto kontrola vlastností prebieha v dvoch režimoch:

- **post-mortem**. Výsledná správa bude zobrazená až po skončení kontroly vlastností,
- **prúdovo**. Priebežné správy pri každom porušení negatívnej vlastnosti na štandardný výstup.

Oba vyššie uvedené režimy a samotná kontrola vlastností a výsledná správa o kontrole sú bližšie popísané v sekcií 3.6.

Procesy diagramu dátových tokov, obrázok 3.1:

- vytvorenie automatu 3.4,
- kontrola vlastností pomocou vytvorených automatov 3.6,
- filtrovanie log súboru na základe vlastností 3.6,
- extrahovanie informácií zo zadaných vlastností pre ladiace účely a pre vytvorenie vstupu pre filter.

3.4 Vytvorenie automatov z vlastností

Vlastnosti sú zadané ako reťazec, čo nie je vhodná reprezentácia pre kontrolu či bola vlastnosť splnená alebo nie. Najlepšou reprezentáciou pre takúto kontrolu sú konečné automaty. Keď je vlastnosť kontrolovaná v log súbore, tak je vyhľadávaná sekvencia udalostí, ktorá buď splní vlastnosť alebo nesplní vlastnosť. Vlastnosť je ako regulárny výraz, ktorý vie prijať určitú vstupnú sekvenciu, ktorá je tomto prípade sekvencia udalostí. Vlastnosť je teda transformovaná na deterministický konečný automat a keď tento automat príde do koncového stavu, nástroj vie, že podmienka je splnená.

3.5 Vlastnosti definované užívateľom

Súbor s vlastnosťami je jedným zo vstupov nástroja pri jeho spustení. Obsahuje všetky informácie potrebné pre konfiguráciu a prípravu nástroja pred filtrovaním log súborov a kontrolou vlastností. V tejto kapitole je podrobne popísaná štruktúra súboru, syntax samostatných častí a syntaktická a lexikálna analýza vlastností.

Súbor s vlastnosťami je vo formáte YAML[1] pretože je to všeobecne rozšírený formát čitateľný človekom a spracovateľný pre väčšinu programovacích jazykov v súčasnosti.

Štruktúra súboru s vlastnosťami

Základná štruktúra súboru je zobrazená v ukážke 3.1. Skladá sa zo štyroch hlavných blokov:

- ***properties* a *badproperties*** majú rovnaký obsah ale rozdielny význam. Ak sa automaty vytvorené z vlastností v bloku *properties* nachádzajú v koncovom stave po skončení kontroly, vlastnosť nebola porušená. V tom istom prípade vlastnosť z bloku *badproperties* bola prorušená. Pokiaľ by sa automat vytvorený z vlastnosti nedostal do koncového stavu, tak vlastnosti v *properties* by boli porušené a vlastnosti v *badproperties* by neboli porušené pretože vlastnosti v *badproperties* sú porušené iba ak automat z takejto vlastnosti skončí v koncovom stave. Vždy sa v súbore musí nachádzať aspoň jeden z týchto dvoch blokov s jednou vlastnosťou. Viac o syntaxi vlastností je napísané v časti 3.5,
- ***events*** je blok obsahujúci udalosti. Viac v sekcií 3.5,
- ***constraints*** je posledný zo základných blokov súboru. Je to zoznam obmedzení, ktoré musia platiť medzi udalosťami ak sa nachádzajú v rovnakej vlastnosti. Napríklad či je zhodný súborový deskriptor pri otvorení a zatvorení súboru.

Príklad obmedzenia (*constraints*). Dve udalosti a otvorenie súboru a b uzatvorenie súboru. Obmedzenie, ktoré musí byť splnené je aby obe udalosti mali rovnaký súborový deskriptor. Popis obmedzenia bude vyzerať $a.1 = b.1$.

Obmedzenia v tejto práci nie sú implementované. Sú však zahrnuté vo vlastnostiach definovaných užívateľom ako vstup nástroja kvôli možnosti rozšíriť túto prácu v budúcnosti. V ladení post-mortem je dôležité kontrolovať nie len či udalosti na seba nadväzujú vo

[1] https://yaml.org/

zvolenej sekvencií (vlastnosti), ale aj či jednotlivé parametre týchto udalostí majú spoločné napr. súborové deskriptory, IP adresy a ďalšie identifikátory.

Popis udalostí

Udalosť v súbore s vlastnosťami je v tvare `kluc: hodnota`, kde `kluc` je identifikátor udalosti. Identifikátor je malé písmeno anglickej abecedy a teda v jednom súbore s vlastnosťami môže byť maximálne 26 udalostí. Druhá z dvojice je `hodnota` a tá popisuje udalosť, ktorá má byť kontrolovaná. Popisovaná je rozšíreným regulárnym výrazom[2] (angl. ERE - extended regular expression).

```
---
properties:
  p1: "grammar for property"
  p2: "grammar for property"
  p3: "grammar for property"

badproperties:
  p4: "grammar for property"

events:
  a: "extended regular expression"
  b: "extended regular expression"
  c: "extended regular expression"

constraints:
  - "grammar for constraints"
  - "grammar for constraints"
...
```

Výpis 3.1: Súbor s vlastnosťami ako vstup nástroja vo formáte YAML.

Gramatika vlastností

Vlastnosť je zjednodušene sekvencia jednej alebo viac udalostí (operandov) a operátorov.

Gramatika pre vlastnosti je definovaná ako štvorica $G = (N, T, P, S)$ [3], kde:

- N sú neterminály `ere, branch, expr, iter`
- T sú terminály `'*', '+', '{', '}', EVENT, n, n1, n2, '(', ')', '|', ','`
- P je konečná množina pravidiel, ktoré sú uvedené vo výpise 3.2
- S je počiatočný neterminál `ere`

Ide o bezkontextovú gramatiku a je to upravená gramatika pre rozšírené regulárne výrazy[2].

[2] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html

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

Výpis 3.2: Pravidlá pre gramatiku vlastností v rozšírenej Bakus-Naur forme (EBNF)

3.6 Monitorovanie vlastností

Táto časť je hlavný proces celého nástroja. Všetko ostatné je len príprava alebo pomocný proces pre monitorovanie vlastností. Proces potrebuje pre svoju činnosť tri dátové štrukrúry:

- vytvorené automaty zo zadaných vlastností,
- informácie o jednom riadku (udalosti) z log súboru, ktorý sa zhoduje s aspoň jednou udalosťou zo súboru vlastností,
- inštancie automatov, ktoré sa posunuli z počiatočného stavu jednou z predchádzajúcich udalostí.

Vytvorené automaty zo zadaných vlastností

Dátová štruktúra uchovávajúca automaty vytvorené z vlastností. Pomocou tejto reprezentácie vlastností monitor kontroluje ich správnosť. Pri každej novej udalosti z log súboru, ktorá sa zhoduje s aspoň jednou udalosťou zadanou v súbore s vlastnosťami sa monitor najskôr pozrie do tejto štruktúry. V každom automate porovnáva možné prechody (udalosti) z počiatočného stavu do ďalšieho možného stavu s udalosťou z log súboru. Inak povedané monitor najskôr kontroluje či môže začať monitorovanie splnenia jednej z vlastností na základe udalosti z log súboru.

Informácie o udalosti z log súboru

Dátová štruktúra s informáciami o udalosti nájdenej v log súbore. Ak táto dátová štruktúra príde na vstup monitoru znamená to, že sa zhoduje s minimálne jednou udalosťou zadanou v súbore s vlastnosťami. Na tomto vstupe monitoru závisí celý beh tohto procesu. Monitor žiada o tento vstup pre každú udalosť z log súboru. Čo znamená, že monitor na tento vstup čaká najdlhšie do skončenia prehľadávania log súboru alebo súborov a keď sa toto prehľadávanie ukončí, tak sa ukončí aj činnosť monitoru.

Informácie, ktoré obsahuje dátová štruktúra:

- identifikátor udalosti pre rozpoznanie v súbore s vlastnosťami,
- udalosť z log súboru,
- číslo riadku, na ktorej bola udalosť nájdená,
- názov súboru, kde bola udalosť nájdená.

Inštancie automatov

Je to zoznam automatov (vlastností), ktoré jedna z predchádzajúcich udalostí posunula z počiatočného stavu do jedného z možných stavov, kde smeruje prechod z počiatočného stavu. To znamená, že ak takáto počiatčná udalosť príde do monitora, tak sa kontroluje či daná vlastnosť bola porušená alebo nie. Každá vlastnosť sa môže kontorolovať toľko krát,

koľko krát prišla do monitora počiatočná udalosť, ktorá naštartovala kontrolu tejto vlastnosti, inak povedané posunula automat z počiatočného stavu.

Inštancia ako element zoznamu neobsahuje iba presnú kópiu automatu (stavy a prechody medzi nimi), ale obsahuje:

- identifikátor automatu (vlastnosti),
- odkaz na popis automatu (stavy a prechody medzi nimi),
- aktuálny stav, v ktorom sa automat nachádza,
- identifikátor vlastnosti aký je v súbore s vlastnosťami,
- informácie o udalostiach z log súborov, ktoré posúvali aktuálny stav na ďalší stav v automate.

Filter log súboru

Filter log súborov je proces hľadania udalostí v log súbore na základe definovaných udalostí rozšírenými regulárnymi výrazmi v súbore s vlastnosťami. Proces monitorovania vlastností žiada tento proces o udalosť a ak filter nájde udalosť odpovedajúcu niektorej udalosti v súbore s vlastnosťami pošle ju ako odpoveď procesu monitorovania vlastností. Čo filter log súboru posiela pri zhode udalostí je popísané v tejto časti 3.6.

Post-mortem analýza

Základ nástroja je poskytnúť používateľovi post-mortem analýzu. Nástroj z log súborov a zadaných vlastností, ktoré má overovať v log súboroch vytvorí výslednú správu 4.5 o výsledkoch analýzy až po skončení celého behu nástroja.

Prúdové spracovanie

Pri návrhu tohto nástroja pre ledenie post-mortem mi napadla myšlienka, že by mohli byť log súbory spracovávané aj prúdovo. A to tak, že počas analýzy sú na štandardný výstup vypisované porušenia vlastností, ktoré sú negatívne. Pre pozitívne vlastnosti takéto prúdové spracovanie nie je možné pretože porušenie pozitívnej vlastnosti vie nástroj určiť až po skončení analýzy všetkých udalostí v log súboroch. Kompletná výsledná správa o analýze udalostí je vytvorená až na konci behu nástroja ako tomu je aj pri post-mortem analýze.

Výhodou prúdového spracovania je možná okamžitá reakcia na porušenie negatívnej vlastnosti. Informácia o takomto porušení je na štandardný výstup vypisovaní vo formáte JSON alebo v textovej podobe čitatelnej pre bežného používateľa.

Výsledná správa o analýze

Výsledná správa o analýze je súbor vo formáte JSON[3]. Tento formát bol vybraný kvôli tomu, že je dobre čitateľný pre používateľa a taktiež s ním vie pracovať široká škála prog-

[3] https://www.json.org/

ramovacích jazykov. Štruktúra výslednej správy je uvedená vo výpise 3.3.

Inštancia automatu a teda aj sekvencia udalostí, ktoré automat prijal sú uvedené vo výslednej správe len v dvoch prípadoch.

Ak je porušená pozitívna vlastnosť a automat vytvorený z tejto vlastnosti nie je po poslednej udalosti z log súboru v koncovom stave. Vtedy je možné zistiť zo sekvencie udalostí, ktorá udalosť bola ako posledná a na základe tejto informácie hľadať príčinu porušenia vlastnosti.

Ak je splnená negatívna vlastnosť a automat vytvorený z tejto vlastnosti sa nachádza v koncovom stave. V tom prípade je k dispozícií sekvencia udalostí vedúca k tomuto porušeniu vlastnosti.

Zvyšné dva prípady, keď je splnená pozitívna vlastnosť a ak je porušená negatívna vlastnosť nemá zmysel uvádzať do výslednej správy. V oboch prípadoch sa jedná o želaný výsledok a nie je dôvod na uchovávanie sekvencie udalostí vedúcich k správenmu chovaniu.

```
{
  'properties': {
    'p1': {
      'property': 'grammar for property',
      'violated': []
    }
  },
  'badproperties': {
    'p3': {
      'property': 'grammar for property',
      'violated': [
        {
          'id': 'automaton instance id',
          'is_property_met': 'true/false',
          'events_sequence': [
            {
              'event_id': 'english alphabet',
              'log_file': 'log file name',
              'log_lineno': 'integer',
              'log_line': 'content of that line'
            }
          ]
        }
      ]
    }
  }
}
```

Výpis 3.3: Vysledná správa po skončení behu programu vo formáte JSON.

Tabuľka 3.1: Špecifikácia požiadaviek

Identifikácia (RQ_XX)	Popis	Kategória	Hierarchia
RQ_00	Program by mal vedieť overiť vlastnosti zadané užívateľom v zadanom log súbore.	Požiadavky na projekt	
RQ_01	Program by mal spracovať vlastnosti definované regulárnou gramatikou. (ERE)	Funkcionálne	RQ_00
RQ_02	Program by mal rozlišovať medzi pozitívnymi a negatívnymi vlastnosťami.	Funkcionálne	
RQ_03	Vstupný log súbor by mal byť spracovávaný prúdovo.	Funkcionálne	
RQ_04	Program bude mať po skončení výslednú správu o porušení alebo splnení vlastností.	Požiadavky na projekt	
RQ_05	Výsledná správa by mala obsahovať stopu pre objavenie pôvodu chyby.	Požiadavky na projekt	RQ_04
RQ_06	Výsledná správa by mala obsahovať identifikátor vlastnosti, ktorá nebola splnená.	Požiadavky na projekt	RQ_04
RQ_07	Výsledná správa o výsledku testovania musí byť čitateľná a prehľadná.	Použiteľnosť	RQ_04
RQ_08	Gramatika pre vlastnosti by mala byť pochopiteľná a ľahko zrozumiteľná.	Použiteľnosť	
RQ_09	Program musí byť multiplatformný.	Požiadavky na projekt	
RQ_10	Log súbor môže byť zadaný ako parameter príkazového riadku alebo na štandardný vstup.	Požiadavky na projekt	
RQ_11	Môže byť zadaných viac log súborov naraz a môžu byť zkomprimované vo formáte gzip.	Požiadavky na projekt	RQ_10

Kapitola 4

Implementácia nástroja pre ladenie post-mortem

V tejto časti sú popísane implementačné detaily nástroja pre ladenie post-mortem. Pre implementáciu bol zvolený programovací jazyk Python 3. Nástroj využíva rozhranie príkazového riadku, ale jednotlivé moduly sú implementované nezávisle na sebe a môžu byť využívané aj samostatne pre iné implementácie.

4.1 Adresárová štruktúra a používanie nástroja

V tejto časti je znázornená adresárová štruktúra 4.1 doplnená o ďalšie informácie a ako sa používa nástroj respektíve aké su jeho možnosti.

Ako sa používa nástroj pre ladenie post-mortem:

- súbor s vlastnosťami musí byť zadaný pri každom spustení nástroja,
- log súbor alebo adresár s log súbormi musí byť zadaný buď ako parameter príkazového riadku alebo na štandardný vstup,
- môže byť špecifikovaná cesta, kde má byť uložená výsledná správa, inak bude uložená v koreňovom adresári nástroja,
- možnosť zapnutia prúdového spracovania, kedy porušené vlastnosti budú vypisované počas spracovania na štandardný výstup v dvoch režimoch.

Presný opis ako sa správne používajú prepínače a aké majú označenie je uvedené v súbore `README.md`. Pre ukážku ako môže vyzerať spustenie nástroja pre ladenie post-mortem ak aktuálny adresár je koreňový adresár nástroja v operačnom systéme Linux:
`python logchecker.py -p tests/properties/syslog.yml -l tests/logs/syslog`

```
./
        docs/
                documentation
        logchecker/
                source files (Python)
        tests/
                inputs/
                        JSON files for testing
                logs/
                        log files
                properties/
                        properties files
                __init__.py
                conftest.py
                test_au_builder.py
                test_prop_parser.py
        .gitignore
        logchecker.py
        properties.yml
        README.md
        requirements.txt
        setup.py
```

Výpis 4.1: Adresárová štruktúra nástroja pre ladenie post-mortem. Znázorňuje obsah koreňového adresára v určitej miere abstrakcie. Detaily sú vysvetlené v texte tejto časti.

V adresárovej štruktúre vo výpise 4.1 sa v zložke `./docs/` nachádzajú požiadavky na projekt, dokumentácia, obrázky príkladov použité v `README.md` a výstupný textový súbor `pasrer.out`, ktorý bol po prvom spustení lexikálnej a syntaktickej analýzy vygenerovaný. Obsahuje detailný opis gramatiky a opis jednotlivých stavov automatu, ktorý analýzu vykonáva.
Zložka `./tests/inputs/` obsahuje vstupné a výstupné hodnoty pre testovanie lexikálnej a syntaktickej analýzy a testovanie, či boli automaty z vlastností vytvorené správne. Zložky `./tests/logs/` a `./tests/properties/` obsahujú vzorky log súborov rôznych služieb alebo programov a súbory s vlastnosťami, ktoré majú byť kontrolované. Ukazujú príklady ako môže byť nástroj použitý.

4.2 Trieda AutomatonBuilder

Trieda implementuje vytvorenie automatu z vlastnosti zadanej v súbore s vlastnosťami. Trieda využíva viacero ďalších tried a sama je zdedená z triedy `BaseBuilder`, čo je trieda, ktorá implementuje základné vytváranie konečného automatu. Kompletný diagram vytvárania konečného automatu je v prílohe B.1.

Ako rozhranie pre vytvorenie konečného automatu je metóda `build`. Trieda vytvára deterministický konečný automat z vlastnosti, ktorá je reprezentovaná abstraktným syntaktickým stromom. Tento abstraktný syntaktický strom je výstupom lexikálnej a syntaktickej

analýzy. Trieda `AutomatonBuilder` implementuje jednotlivé operátory spájajúce udalosti (vlastnosť) ako metódy.

Proces vytvorenia nedeterministického konečného automatu s epsilon prechodmi je rekurzívny. Postupne sa vytárajú čiastkové automaty, ktoré sa postupne spájajú od najnižších úrovní abstraktného syntaktického stromu a na záver je vytvorený operátor v koreňovom uzle abstraktného syntaktického stromu. Pokiaľ sa v koreňovom uzle nenachádza operátor, ale udalosť, tak je vlastnosť iba jediná udalosť a vytvára sa automat iba pre túto jedinú udalosť. Výsledný nedeterministický konečný automat s epsilon prechdomi je reprezentovaný triedou `Automat`, čiže každý automat je objekt. Na vytvorenie deterministického konečného automatu som využil balík tretej strany *automata-lib*[1]. Dokáže vytvoriť z nedeterministického konečného automatu s epsilon prechdomi deterministický konečný automat. Má aj svoju reprezentáciu deterministického konečného automatu, ktorá je ďalej používaná pri monitorovaní vlastností.

Reprezentácia nedeterministického konečného automatu mohla byť už v tvare, ktorý má balík *automata-lib*, ale z hľadiska čitateľnosti a pochopenia zdrojového kódu mi vlastná reprezentácia prišla praktickejšia.

4.3 Trieda PropertyParser

Dôležitou súčasťou nástroja je lexikálna a syntaktická analýza vlastností. Ako je popísané v návhru nástroja, u vlastností ide o bezkontextovú gramatiku nakoľko vychádza z gramatiky rozšírených regulárnych výrazov. Vlastná implementácia lexikálnej a syntaktickej analýzy by bola neefektívna a pri dnešných možnostiach zbytočná. Kvôli spomentým dôvodom som pre implementáciu použil balík tretej strany *ply*[2]. Tento balík implementuje nástroje Lex a Yacc.

Implementácia lexikálnej analýzy spočíva iba v definovaní terminálov, čo zahŕňa definovanie názvu a definovanie čo má názov reprezentovať regulárnym výrazom. Prípadne pridaním metódy definovať ďalšie operácie s terminálom pred jeho ďalším použitím. Napríklad ak sa jedná o celé číslo je potrebné konvertovať premennú na príslušný dátový typ, pretože implicitne je to reťazec.

Syntaktická analýza

Pre syntaktickú analýzu stačí pre každý neterminál vytvoriť metódu, ktorej stačí zadať pravidlá, ktorých ľavá strana je práve ten neterminál. Sú to pravidlá vo výpise 3.5, avšak musia byť v Bakus-Naurovej forme (BNF).

Dôležité pri syntaktickej analýze bolo vytvoriť abstraktný syntaktický strom, ktorý je výstupom analýzy a ako ďalšia reprezentácia vlastnosti pre jej nasledujúce použitie. Pri rozhodovaní ako implementovať abstraktný syntaktický strom som bral do úvahy v prvom rade jednoduchosť a možnosť zanorovať túto štruktúru a taktiež ľahkú manipuláciu pri imeplementácií ďalších vecí. Použil som pole, ktoré je v jazyku Python dátová štruktúra `list` a vždy je to pole obsahujúce maximálne dva prvky až na jednu výnimku. Prvý prvok poľa je operátor a druhý prvok je operand (reťazec) alebo viac operandov (pole). Prvý prvok je koreňový uzol abstraktného syntaktického stromu a druhý provk je jeden alebo viac

[1] https://pypi.org/project/automata-lib/
[2] https://www.dabeaz.com/ply/

```
property_1 = 'a'
ast_1 = 'a'

property_2 = 'b*'
ast_2 = ['*', 'b']

property_3 = 'ab|c'
ast_3 = ['|', [['', ['a', 'b']], 'c']]

property_4 = '(ab+)*'
ast_4 = ['*', ['', ['a', ['+', 'b']]]]

property_5 = '(a(b|c)){2}'
ast_5 = ['{2}', ['', ['a', ['|', ['b', 'c']]]]]
```

Výpis 4.2: Ukážka vlastností ako je zadaná užívateľom a ako je reprezentovaný po lexikálnej a syntaktickej analýze abstraktný syntaktický strom v implementácií

priamych nasledovníkov koreňového uzlu. Operandy môžu byť ďalšie zanorené operátory a operandy takže hĺbka stromu môže logicky narastať. Pre ukážku ako je implementovaný abstraktný syntaktický strom sú príklady v časti 4.2.

Vyššie spomínaná výnimka je jediný prípad kedy vlastnosť je iba jedna udalosť (operand) t.j. abstraktný syntaktický strom má iba koreňový uzol a stačí ak je reprezentovaný ako reťazec kedže sa jedná iba o identifikátor udalosti.

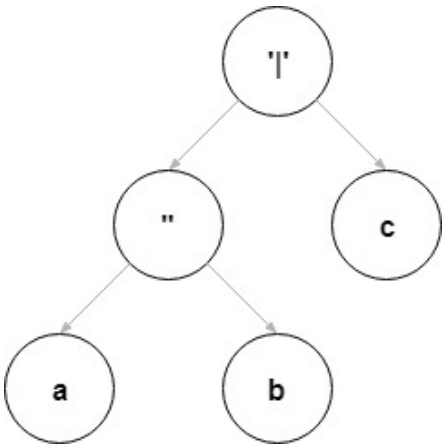
Príklady abstraktného syntaktického stromu

Základná dátová štruktúra, ktorá uchováva uzol a jeho priameho nasledovníka `[operator, operand]` alebo priamych nasledovníkov `[operator, [operandy]]`.

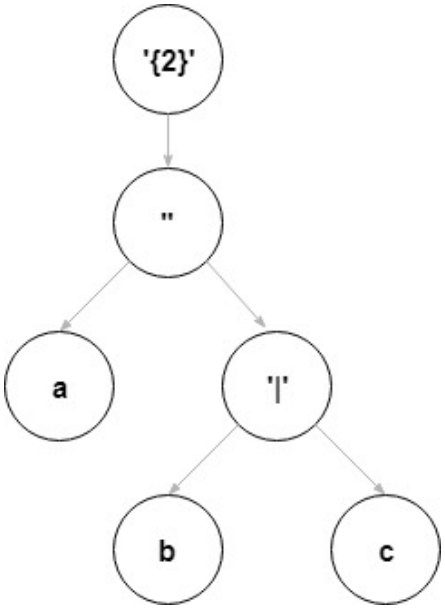
Ukážka niekoľkých možností ako môže potenciálne vyzerať mňou zvolená reprezentácia vlastnosti ako abstraktného syntaktického stromu v implementácií a vizuálna ukážka pre lepšie pochopenie na obrázkoch 4.1 a 4.2.

4.4 Trieda Monitor

Hlavná trieda programu, ktorá implementuje kontrolu či boli vlastnosti porušené alebo nie. Kontrola je zahájená invokovaním metódy `monitoring`. Prebieha v nekonečnom cykle, kde si metóda iteratívne volá metódu z triedy `LogFilter` pre nájdenie udalosti. Pokiaľ sa udalosť z log súboru zhoduje s aspoň jednou udalosťou definovanou v súbore s vlastnosťami, nástroj spustí kontrolu nad už overovanými vlastnosťami alebo prípadne začne overovať nejakú vlastnosť od začiatku. Hľadanie udalosti v log súbore a aj nekonečný cyklus kontroly končí po poslednej udalosti v log súbore.



Obr. 4.1: Diagram dátových tokov - Súčinnosť jednotlivých častí nástroja



Obr. 4.2: Diagram dátových tokov - Súčinnosť jednotlivých častí nástroja

Ak sa našla udalosť odpovedajúca udalosti zo súboru s vlastnosťami, tak sa pre túto udalosť alebo viac udalostí invokuje metóda `read_event`. Metóda nájdenú udalosť dostane ako parameter a kontroluje či je možný posun v už rozpracovanej kontrole nejakej vlastnosti o túto udalosť alebo je možnosť rozpracovať novú kontrolu nejakej vlastnosti, ktorú spustí nájdená udalosť v log súbore. Pri tejto kontrole metóda overuje, či jeden z automatov sa nachádza v koncovom stave.

Pri kontrole ako je uvedené v návrhu nástroja je možné vyhodnotiť iba negatívnu vlastnosť a teda do výslednej správy je možné počas kontroly zahŕňať iba sekvencie udalostí, ktoré splnili negatívnu vlastnosť. Pozitívne vlastnosti sú vyhodnocované až po konci kontroly a sú uvedené sekvencie udalostí, ktoré nedoviedli rozpracované automaty do koncového stavu. To znamená, že výsledná správa sa z časti vytvára počas kontroly a zvyšok až po skončení kontroly.

Sekvencie udalostí, ktoré splnili negatívnu udalosť a teda doviedli automat do koncového stavu je možné vypisovať na štandardný výstup hneď pri zistení, že túto negatívnu vlastnosť splnili.

Trieda LogFilter

Trieda pracujúca s log súbormi. Na prehľadávanie súborov používa modul *fileinput* zo štandardnej Python knižnice. Tento modul vytvorí objekt, ktorý prehľadáva zvolený súbor alebo adresár alebo je možné zadať adresár alebo súbor na štandardný vstup. Dokáže prehľadávať aj log súbory archivované rotáciou log súborov, ktorá je používaná pre zmenšenie veľkosti týchto súborov. Vytvorený objekt vracia riadok po riadku a ku každému riadku sú o ňom informácie a z tých sa vytvára dátová štruktúra 3.6, ktorú potrebuje trieda `Monitor` pre svoju činnosť. Trieda `LogFilter` funguje ako generátor. Generátor umožňuje získavať riadok po

riadku bez toho aby sa obsah log súborov načítal do pamäti a zabraňujem tak zbytočnému zahlteniu pamäte, kde pri práci s jedným riadkom mám v pamäti informácie iba o ňom.

Riadok alebo udalosť práve načítaná v pamäti je porovnávaná s rozšírenými regulárnymi výrazmi, ktoré sú definované v súbore s vlastnosťami. Pri každom riadku z log súboru sa porovnáva so všetkými regulárnymi výrazmi a môže sa zhodovať so žiadnou, jednou alebo aj viac zadanými udalosťami užívateľom. V najhoršom možnom prípade môže byť v súbore s vlastnosťami definovaných až 26 udalostí viď. 3.5 a každý riadok bude porovnávaný s 26 regulárnymi výrazmi čo je časovo náročná operácia pri veľkom počte riadkov v log súbore. Efektívnejší spôsob možný v jazyku Python by bolo spojiť všetky regulárne výrazy do jedného a zistiť pri zhode, s ktorou časťou sa riadok zhodoval a určiť identifikátor udalosti. No spomínaný spôsob nie je možný pretože riadok sa môže teoreticky zhodovať s viacerými udalosťami ako už bolo uvedené a efektívnejší spôsob by nesplnil túto požiadavku.

4.5 Testovanie

V tejto časti je popísané testovanie nástroja ako celku, teda od zadania vlastností a udalostí v súbore s vlastnosťami a zadania log súborov alebo súborov pre kontrolu až po výslednú správu. A rovnako aj testovanie niektorých častí nástroja ako separátne celky.

Testovanie samostatných častí nástroja je implementované pomocou balíka *pytest*. Je vhodný na testovanie rozhraní, čo v prípade nástroja pre ladenie post-mortem, ktorého časti sú implementované nezávisle od seba a fungujú spolu cez svoje rozhrania je ideálny balík pre testovanie.

Testovanie syntaktickej a lexikálnej analýzy vlastností

Testovanie syntaktickej a lexikálnej analýzy spočíva v kontrole reálneho výstupu analýzy voči očakávanému výstupu. Očakávaný výstup je abstraktný syntaktický strom. Jeho reprezentácia je popísaná tu 4.3. Vstup analýzy je vlastnosť podľa definovaných pravidiel 3.5. Tieto testy sú implementované v súbore `tests/test_prop_parser.py`.

Overenie správnosti výstupu z určitého vstupu lexikálnej a syntaktickej analýzy je rovnaké ako používanie analýzy v implementácií nástroja. Ide o volanie metód `PropertyParser.run(property)` a následne `PropertyParser.get_ast()`. Vstup a očakávaný výstup sú zadané v súbore `tests/inputs/prop_parser.json`, z ktorého si testovací balík načíta hodnoty a ďalej s nimi pracuje.

Vstupy (vlastnosti) som zvolil tak, aby každá operácia bola testovaná separátne a potom aj ich kombinácie a rôzne zanorenia.

Testovanie automatov vytvorených z vlastností

Overiť správnosť vytvorených automatov z vlastností je možné iba ak je automatu na vstup zadaná sekvencia udalostí a táto sekvencia zodpovedá zadanej vlastnosti podľa gra-

matiky 3.5 a sekvenciu udalostí vie automat prijať.

Spôsob ako implementovať takéto testovanie je jednoduchý. Balík tretej strany *automata-lib* vytvárajúci deterministický konečný automat ponúka vo svojom rozhraní overenie, či je zadaná vstupná sekvencia automatom akceptovaná alebo nie. Zostáva už iba vygenerovať vstupnú sekvenciu, ktorá by zodpovedala vlastnosti (automatu). Ako bolo spomenuté v predošlom odseku, gramatika pre vlastnosti je iba obmedzená gramatika pre rozšírené regulárne výrazy, z čoho vyplýva, že s vlastnosťou môžem pracovať ako s rozšíreným regulárnym výrazom. To mi umožňuje využiť balík tretej strany *rstr*[3] generujúci náhodné sekvencie napríklad z regulárnych výrazov.

Testovanie prebieha v súbore `tests/test_au_builder.py` tak, že sa zo zadanej vlastnosti v súbore `test/inputs/au_builder.json` vytvorí deterministický konečný automat a vygeneruje sa náhodná vstupná sekvencia a overí sa, či je vstupná sekvencia akceptovaná automatom alebo nie.

Overovanie správnosti nástroja na log súboroch

Najpodstatnejšie je testovanie nástroja ako celku, tak ako má fungovať pri používaní. K tomuto účelu som vytvoril niekoľko vzoriek log súborov z rôznych služieb a stôp jednoduchých programov v adresári `tests/logs/`. Veľmi mi pomohli vzorky log súborov z dokumentácie nástroja OSSEC[4]. A ešte som vytvoril ukážkové súbory s vlastnosťami pre tieto log súbory v adresári `tests/properties/`. Vlastnsti overujú základné udalosti, ktoré by mali byť splnené alebo naopak tie, ktoré môžu viesť k nejakej chybe.

Príklady priložené spolu spolu s testovaním a aké vlastnosti v nich boli overované:

- stopa programu - program, ktorý je ako ukážka rekurzie a musí byť splnená vlastnosť, že funkcia nebude rekurzívne zanorená viac krát ako bolo zadané v programe,
- log súbor o inštalácií napr. programov v operačnom systéme - musí byť splnená vlastnosť či je nainštalované všetko potrebné pre programovanie v jazyku Python a vlastnosť, či je nainštalovaný aspoň jeden program pre úpravu zdrojového kódu,
- syslog - kontrola, či DHCP požiadavka z konkrétnej IP adresy dostala aj DHCP potvrdenie,
- záznam z Windows firewall - splnenie pozitívnej vlastnosti, či otvorená komunikácia bola aj ukončená a negatívna vlastnosť, či nebol prijatý datagram z IP adresy, ktorý mal byť zahodený.

Príklad spustenia nástroja spolu so súborom s vlastnosťami 4.4 a log súborom 4.3 z programu Windows fierwall a výsledná správa o kontrole vlastností 4.5:
Spustenie nástroja v prostredí Linux, v koreňovom adresári projektu:

```
python logchecker.py -p win_fierwall.yml -l win_fierwall
```

[3] https://pypi.org/project/rstr/
[4] https://ossec-docs.readthedocs.io/en/latest/log_samples/

```
2006-09-19 03:40:30 OPEN UDP 192.168.72.12 10.20.72.186 3702 389 - - - - - - - - -
2006-09-19 03:40:54 OPEN UDP 192.168.183.114 239.255.255.250 65169 1900 310 - - - - - - - RECEIVE
2006-09-19 03:40:54 DROP UDP 192.168.183.114 239.255.255.250 65169 1900 310 - - - - - - - RECEIVE
2006-09-19 03:40:54 DROP UDP 192.168.183.114 239.255.255.250 65168 1900 319 - - - - - - - RECEIVE
2006-09-19 03:40:54 DROP UDP 192.168.183.114 239.255.255.250 65168 1900 319 - - - - - - - RECEIVE
2006-09-19 03:41:15 DROP UDP 172.20.73.241 239.255.255.250 2250 1900 250 - - - - - - - RECEIVE
2006-09-19 03:37:57 OPEN TCP 192.168.72.12 10.20.158.58 3686 80 - - - - - - - - -
2006-09-19 03:37:57 CLOSE TCP 192.168.72.12 10.20.158.58 3686 80 - - - - - - - - -
2006-09-19 03:37:57 OPEN TCP 192.168.72.12 10.20.158.58 3687 80 - - - - - - - - -
2006-09-19 03:37:57 CLOSE TCP 192.168.72.12 10.20.158.58 3687 80 - - - - - - - - -
2006-09-19 03:37:57 OPEN TCP 192.168.72.12 10.20.158.58 3688 80 - - - - - - - - -
2006-09-19 03:37:57 CLOSE TCP 192.168.72.12 10.20.158.58 3688 80 - - - - - - - - -
```

Obr. 4.3: Vzorka log súboru z programu Windows firewall nad ktorým sú kontrolované vlastnosti

```yaml
---
# log from windows firewall

properties:
  # checking if opened connection was closed
  p1: "(de)*"

badproperties:
  # checking if UDP packet which should be dropped was accepted
  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.*$"

constraints:
  - "grammar for constraints"
...
```

Obr. 4.4: Súbor s vlastnosťami vo formáte YAML. Pozitívna vlastnosť kontroluje, či otvorená TCP komunikácia bola aj ukončená a negatívna vlastnosť kontroluje či náhodou nebol prijatý UDP datagram, ktorý má byť zahodený.

Obr. 4.5: Výsledná správa vo formáte JSON. Vieme z nej zistiť, že jeden datagram nebol zahodený, čo by sa nemalo stať.

Kapitola 5

Záver

Cieľom práce bolo analyzovať rôzne stopy programov a užívateľovi umožniť overovať nad nimi rôzne vlastnosti, ktoré si špecifikuje a tak mu uľahčiť ladenie po skončení behu programu. Nástroj bol vytvorený tak, že užívateľ si zadá aké vlastnosti chce overovať a na akých stopách programu a výsledom bude správa či vlastnosti boli porušené alebo nie. Či už ide o pozitívne vlastnosti alebo negatívne, ktoré ak sú splnené poukazujú chybu.

Návrh práce bol založený na tom aby vlastnosti zadávané užívateľom nebolo problém pre užívateľa vytvoriť a nezdržoval sa ich vytváraním a zároveň aby bola zvolená taká reprezentácia, ktorej by mal každý kto bude nástroj používať rozumieť. Testovanie bolo zamerané na kontrolu rôznych vlatsností na rôznych vzorkách log súborov a na správne spracovanie vlastnosti pre samotnú analýzu log súborov.

Táto práca mi rozšírila vedomosti v oblasti ladenia programov a utvrdila ma v dôležitosti venovať sa oblasti testovania a jej automatizácií pre skrátenie časovej a finančnej náročnosti bez vplyvu na menšiu kvalitu testovania a nasmerovala ma na cestu, ktorou sa chcem v ďalšom vývoji uberať.

Nástroj nie je integrovaný pre platformu Testos. Medzi obmedzenia patrí chýbajúca implementácia obmedzení, ktoré vlastnosti musia spĺňať. Ide o rovnosť parametrov medzi dvomi udalosťami vo vlastnosti, za účelom či na seba tieto udalosti nadväzujú. Toto je však možné až po rozšírení nástroja o vlastnosti obsahujúce parametre.

Okrem rozšírenia nástroja o vlastnosti s parametrami je ďalšie pokračovanie na vývoji tohto nástroja analýza viac vláknových programov. Taktiež idea analýzy v reálnom čase s čím by bola možná reakcia na problémy takmer okamžite v bežiacich systémoch.

Literatúra

[1] *Micro Focus Security ArcSight Common Event Format.* [Online; navštíveno 04.04.2019].
URL https://community.microfocus.com/t5/ArcSight-Connectors/ArcSight-Common-Event-Format-CEF-Implementation-Standard/ta-p/1645557?attachment-id=68077

[2] *SIEM market, log management tools need a standardized log format.* [Online; navštíveno 04.04.2019].
URL https://searchsecurity.techtarget.com/magazineContent/SIEM-market-log-management-tools-need-a-standardized-log-format

[3] Alexander Meduna, R. L.: *Formální jazyky a překladače IFJ - Studijní opora.* [Online; navštíveno 04.04.2019].
URL https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIFJ-IT%2Ftexts%2FOporaIFJ.pdf&cid=12153

[4] Gutenev, A.: *Post-mortem Debugging of Windows Applications.* [Online; navštíveno 04.04.2019].
URL https://www.slideshare.net/GlobalLogicUkraine/postmortem-debugging-of-windows-applications

[5] John Levine, T. M.; Brown, D.: *lex & yacc.* O'Reilly & Associates, Inc., 1992, ISBN 1-56592-000-7.

[6] Karen Kent, M. S.: *Guide to Computer Security Log Management.* [Online; navštíveno 02.04.2019].
URL https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-92.pdf

[7] Klein, A.: *The power of post-mortem debugging.* [Online; navštíveno 04.04.2019].
URL https://almarklein.org/pm-debugging.html

[8] Marty, R.: *Open Log Format – What a Great Standard – Not.* [Online; navštíveno 04.04.2019].
URL https://raffy.ch/blog/2007/09/14/open-log-format-what-a-great-standard-not/

[9] Reger, G.; Havelund, K.: What is a Trace? A Runtime Verification Perspective. In *ISoLA 2016: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, Lecture Notes in Computer Science, 2016, s. 339–355, doi:10.1007/978-3-319-47169-3_25.

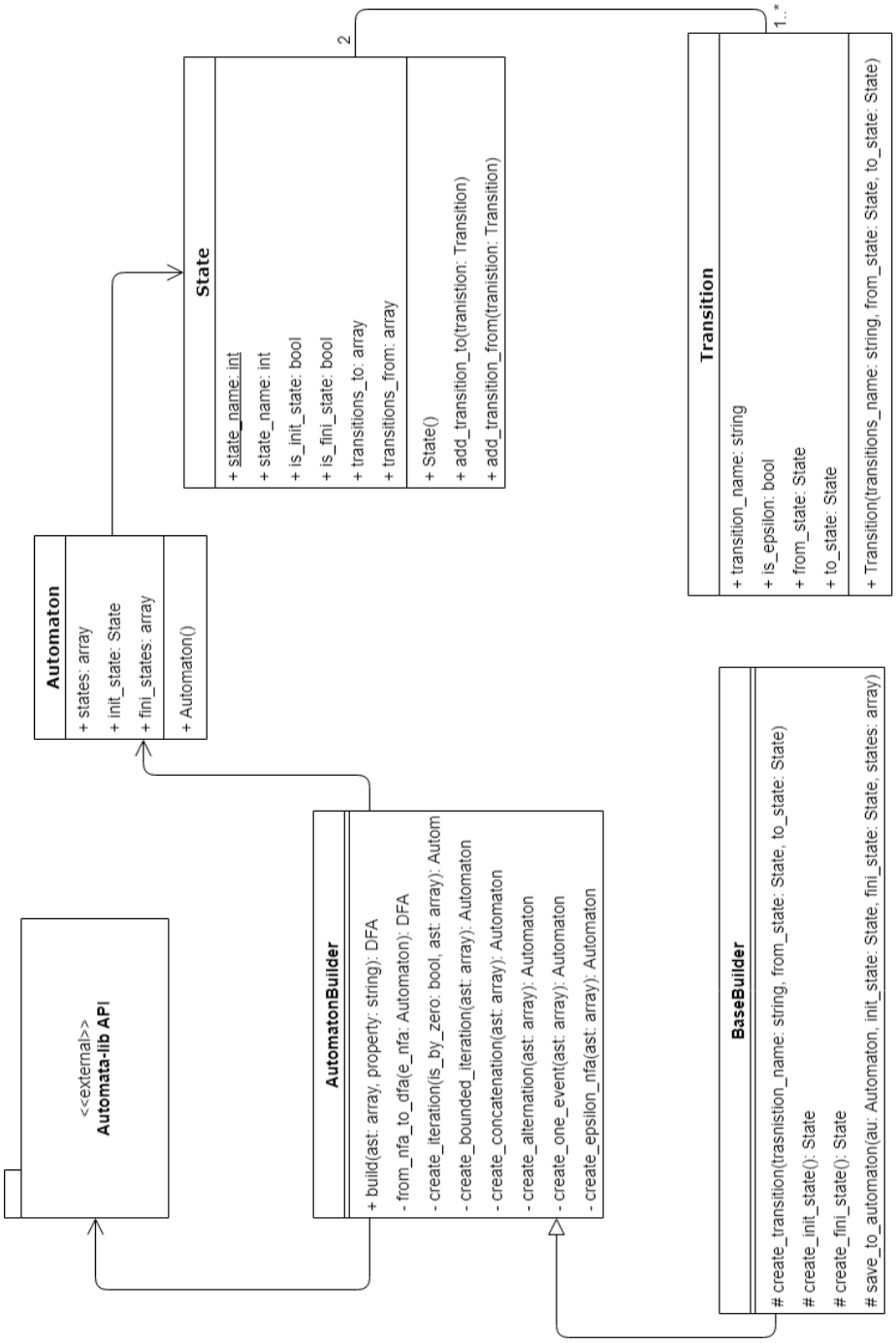
[10] Rosa, A. L.: *Log Monitoring: What should we do before we start?* [Online; navštíveno 12.04.2019].
URL https://blog.pandorafms.org/log-monitoring/

Príloha A

Médium so zdrojovými súbormi

Príloha B

Diagram tried vytvorenia automatu z vlastnosti



Obr. B.1: Diagram tried vytvárania deterministického konečného automatu z vlastnosti zadanej v log súbore.