VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

GENERÁTOR TESTOVACÍCH BEHOV NAD GUI

GENERATOR OF TEST RUNS OVER GUI

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. JURAJ SOJČÁK**
AUTHOR

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

BRNO 2018

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Sojčák Juraj, Bc.**
Program: Informační technologie Obor: Inteligentní systémy
Název: **Generátor testovacích běhů nad GUI**
Generator of Test Runs over GUI
Kategorie: Analýza a testování softwaru
Zadání:

1. Nastudujte testování založené na modelech systému. Nastudujte principy automatizovaného generování testovacích sad. Zaměřte se na návrh testovacích běhů. Nastudujte testování grafického uživatelského rozhraní.
2. Navrhněte nástroj pro vyhledávání cest v modelu testované aplikace podle zadaného kritéria pokrytí grafických uživatelských rozhraní. Kritéria pokrytí by měla být definovatelná uživatelem. Výstupem nástroje bude sada testovacích běhů.
3. Integrujte nástroj jako podsystém platformy Testos.
4. Ověřte funkcionalitu pomocí automatizovaných jednotkových a integračních testů. Demonstrujte funkcionalitu nástroje na sadě jednoduchých případů použití GUI testované aplikace.

Literatura:

- D. Xu, W. Xu, M. Kent, L. Thomas and L. Wang, "An Automated Test Generation Technique for Software Quality Assurance," in *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 247-268, March 2015. doi: 10.1109/TR.2014.2354172
- D. Xu, W. Xu and M. Tu, "Automated Generation of Integration Test Sequences from Logical Contracts," *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, Vasteras, 2014, pp. 632-637. doi: 10.1109/COMPSACW.2014.106

Při obhajobě semestrální části projektu 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: 22. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Grafické používateľské rozhrania tvoria významnú časť počítačových systémov. Z tohto dôvodu sú kladené čoraz väčšie nároky na ich správne fungovanie, ktoré môže byť dosiahnuté testovaním. Táto práca popisuje princíp testovania na základe modelu, pomocou ktorého je možné automatizovane generovať testovacie prípady. V tejto práci je definovaný model, ktorý je možné využiť na modelovanie bežne používaných GUI, algoritmus na získavanie takéhoto modelu z GUI a prostriedok na definovanie kritéria pokrytia modelu GUI.

Abstract

Graphical user interfaces represent significant part of computer systems. This causes an increase of demands for their proper behavior, which can be achieved by a proper testing during development. This thesis describes the principles of model based testing for automated generation of test cases. In this way, the productivity of test teams can increase. The thesis defines model for modeling the most common GUI elements, algorithm for obtaining such a model from the GUI, and a tool for definition of coverage criteria.

Klíčová slova

testovanie GUI, testovanie na základe modelu, MBT

Keywords

GUI testing, model-based testing, MBT

Citace

SOJČÁK, Juraj. *Generátor testovacích behov nad GUI*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Generátor testovacích behov nad GUI

Prohlášení

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

.....
Juraj Sojčák
22. května 2019

Poděkování

Ďakujem môjmu vedúcemu Ing. Alešovi Smrčkovi, Ph.D. za jeho pomoc a odborné rady poskytnuté pri písaní tejto práce.

Obsah

Kapitola 1

Úvod

Používateľské rozhranie predstavuje súbor techník a mechanizmov určených na interakciu človeka so systémom. Jedným z typov používateľských rozhraní je grafické používateľské rozhranie označované ako GUI. Prvé zmienky o GUI sa spájajú s počítačom *Xerox Alto*, ktorý vznikol v roku 1973, v ktorom sa prvý krát objavilo grafické používateľské rozhranie, také ako poznáme dnes a označovalo sa *WIMP*. V súčasnosti sa stretávame s GUI v rôznych formách a na rôznych platformách, najmä kvôli jeho širokému spektru použitia.

Spolu s GUI sa postupom času vyvíjalo aj softvérové inžinierstvo a s tým spojené testovanie softvéru. Proces testovania sa snaží systematicky pristupovať k odhaľovaniu a nahlasovaniu chýb vo vyvíjaných systémoch. Takto prispieva ku zvyšovaniu kvality softvéru a zároveň aj k samotnému procesu vývoja softvéru. Existuje niekoľko rôznych techník ako pristupovať k testovaniu GUI. Medzi najrozšírenejšie techniky patria manuálne testovanie a automatizované testovanie. Pri automatizovanom testovaní testeri vytvárajú skripty, testy, ktoré je možné automatizovane spúšťať. Tento proces síce prináša možnosť regresného testovania, ale vytváranie jednotlivých testovacích prípadov môže byť zdĺhavé. Navyše je nutné pri každej zmene GUI testy aktualizovať, čo je náročné pri rozsiahlych aplikáciách. Testovanie na základe modelu (angl. *model-based testing – MBT*) generuje testovacie prípady automaticky. Pri tomto prístupe k testovaniu sú testy vytvorené na základe abstraktného popisu systému vo forme modelu. Pri generovaní sa musia definovať požiadavky, ktoré musia testy spĺňať vzhľadom k modelu a teda aj k testovanému systému.

V kapitole 2 je všeobecne popísaný proces testovania spolu s popisom zavedených pojmov požívaných pri testovaní. Kapitola sa následne venuje popisu najznámejších techník testovania GUI. Ďalšia kapitola 3 približuje testovanie na základe modelu a sú v nej popísané a zhodnotené existujúce nástroje, ktoré je možné používať pre testovanie GUI a využívajú princíp MBT. V kapitole 4 sú definované požiadavky pre nástroj a konceptuálny návrh na systém pre automatické testovanie GUI pomocou MBT, a sú tu popísané jednotlivé časti pomocou štrukturálneho a behaviorálneho návrhu. Implementačné detaily systému sú popísane v kapitole 5 a kapitola 6 sa venuje testovaniu a demonštrácii implementovaného nástroja.

Kapitola 2

Testovanie grafických používateľských rozhraní

Grafické používateľské rozhrania sú v porovnaní s iným typom používateľských rozhraní jednoduchšie a intuitívnejšie na ovládanie, kvôli čomu sa rýchlo rozšírili na rôzne platformy od osobných počítačov až po vstavané systémy. Dnešné GUI vzniklo na základe rozhrania *WIMP*, ktoré sa skladá zo štyroch základných komponent – okná (angl. *windows*), ikony (angl. *icons*), navigácia (angl. *menus*), polohovacie zariadene (angl. *pointing device*) [17].

Na rozdiel od práce s GUI z používateľského hľadiska, je testovanie GUI náročnejší proces. Testovanie GUI spočíva v tom, že každá udalosť vykonaná používateľom v systéme, dostáva systém do nového stavu. Keďže je náročné zosumarizovať a skontrolovať všetky stavy, ktoré môžu v systéme nastať, je prakticky nemožné systém prehlásiť za bezchybný [8].

2.1 Testovanie softvéru

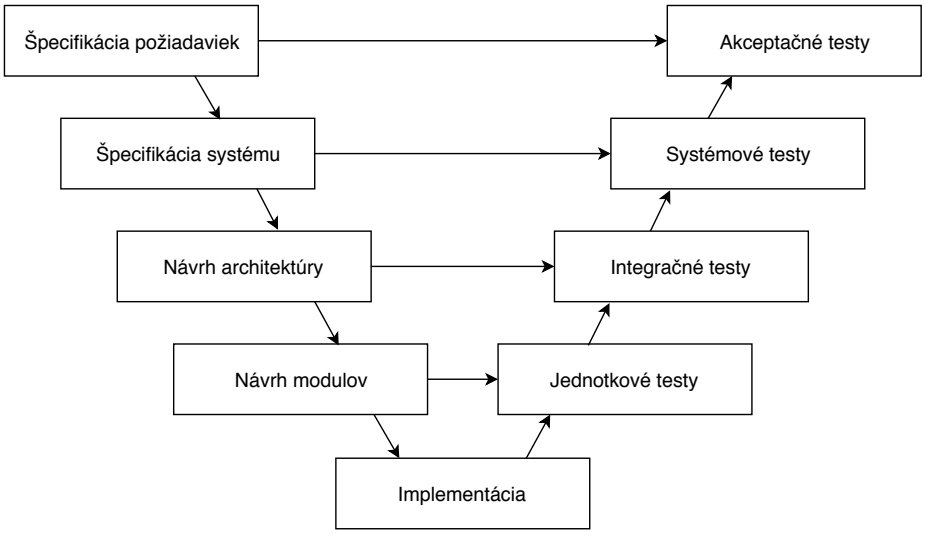
Testovanie softvéru je neoddeliteľnou súčasťou vývoja softvéru. Príkladom môže byť V-model, znázornený na obrázku 2.1, ktorý naznačuje vzťah medzi jednotlivými fázami vývoja softvéru. Testovanie je možné definovať ako systematické preskúmavanie softvéru za účelom odhalenia a nahlásenia chyby. Chyby by mali byť zdokumentované a predané vývojovému tímu na opravu [9].

Okrem jednotlivých úrovní testovania podľa V-modelu, môžeme testovanie rozdeliť na statické a dynamické. Pri statickom testovaní sa skúma softvér bez spúšťania, napríklad. pomocou inšpekcie kódu človekom alebo statickou analýzou jednotlivých príkazov a riadiacich konštrukcií v kóde. Naopak dynamická analýza skúma softvér za behu [9]. Za celú dobu vývoja softvérového inžinierstva bola snaha robiť pokroky najmä v automatizovanom testovaní. Z tohto dôvodu boli rozvíjané viaceré prístupy pre testovanie, napr. testovanie na základe požiadavkov, regresné testovanie alebo testovanie na základe modelu. Testovaním na základe modelu sa zaoberá táto práca.

Nezávisle od procesu, akým je testovanie vykonávané, alebo fázy vývoja softvéru, sa s testovaním spájajú špecifické pojmy, ktoré budú používané aj v tejto práci. Systém, ktorý je cieľom testovania sa v angličtine označuje *System Under Test (SUT)*. Podobne sa môže označovať aj aplikácia (*AUT*), popr. menšia časť aplikácie, ktorá je podrobená testovaniu [1].

Testovací prípad (angl. *test case*) je množina vstupných a výstupných podmienok (angl. *precondition*[1] a *postcondition*[2]), vstupov, jednotlivých krokov, ktoré sa majú vykonať, očakávaných výstupov vzhľadom k SUT. Množina testovacích prípadov sa označuje ako testovacia sada (angl. *test suite*) [1]. Jednotlivé testovacie sady sa navrhujú s cieľom pokrytia určitej časti SUT, kde pokrytie systému môže byť definované určitým kritériom pokrytia (angl. *Coverage Criteria*). Kritériá pokrytia pomáhajú určovať rozsah testovania a väčšinou sa týkajú jednotlivých častí kódu (napr. pokrytie riadkov, pokrytie logických výrazov v podmienených príkazoch, ...) [1].

Pre väčšinu programovacích jazykov existujú sady nástrojov (označované ako *xUnit rámce*[3]), ktoré poskytujú štandardizovaný prístup k vytváraniu testovacích sád, dokážu tieto testovacie sady spúšťať a následne aj vyhodnocovať výsledky [1]. K vyhodnocovaniu výsledkov testov sa používa tzv. testovacie orákulum – mechanizmus, ktorý udáva či test prešiel alebo zlyhal.



Obrázek 2.1: Jeden z najzákladnejších prístupov pri vývoji software je V-model. Zobrazuje jednotlivé fázy vývoja softvéru a s nimi súvisiace fázy testovania. Už v počiatočných fázach je potrebné okrem návrhu systému vytvoriť aj testovacie návrhy a plány, aby v priebehu implementácie systému bolo možné implementovať aj jednotlivé testovacie procesy [19].

2.2 Metódy testovania GUI

V tejto časti budú popísané najpoužívanejšie metódy testovania GUI. Sú tu popísané princípy, výhody a nevýhody jednotlivých prístupov.

Prvou metódou je manuálne testovanie, ktoré vykonávajú ľudia bez automatizácie. Ďalšie prístupy využívajú možnosti nahradenia ľudskej entity pri používaní GUI programovým prostriedkom na ovládanie GUI a získavanie informácií s GUI. Tieto prostriedky dokážu identifikovať ovládacie prvky na obrazovke a simulovať ich používateľské akcie.

[1] Množina podmienok, ktoré musí SUT spĺňať, aby bolo možné spustiť testovací prípad.
[2] Množina podmienok, ktoré musí SUT spĺňať po dokončení testovacieho prípadu.
[3] http://xunitpatterns.com/

2.2.1 Manuálne testovanie

Manuálne testovanie je používané najmä pri počiatočnom preskúmavaní systému. Táto metóda je najstaršia, no napriek tomu je stále používaná, najmä pre svoju jednoduchosť. Manuálne testovanie môžu využívať testeri, ktorí sú súčasťou vývojového tímu, prípadne bežní používatelia, ktorí budú systém využívať (testovanie použiteľnosti, beta testovanie).

Tester manuálne prechádza navrhnuté testovacie prípady krok po kroku a manuálne porovnáva aktuálne výstupy systému s očakávanými výstupmi, ktoré môžu byť vo forme požiadaviek alebo princípov, ktoré musí používateľské rozhranie spĺňať.

Testovanie použiteľnosti vykonávajú používatelia, u ktorých sa predpokladá, že budú systém v budúcnosti používať. Výsledky testovania sú zaznamenávané a vyhodnocované testovacím tímom s cieľom vylepšiť nedostatky systému. Pri testovaní používatelia dostanú sadu úloh, ktoré sa pokúsia vykonať a následne sú zaznamenávané rôzne aspekty, na ktoré používatelia narazili pri vykonávaní týchto úloh.

Nevýhodou manuálneho testovania je najmä časová náročnosť a teda aj vysoká cena. Okrem časovej náročnosti môžu byť výsledky testovania závislé aj na konkrétnom testerovi, ktorý môže nesprávne vyhodnotiť chyby v systéme. Ďalšou nevýhodou je, že manuálne vykonávanie testovacích prípadov a vyhodnocovanie chýb je stereotypná činnosť, čo môže byť pre niekoho frustrujúce. Na druhú stranu je možné odhaliť aj chyby, ktoré by automatizované prístupy neodhalili.

2.2.2 Testovanie zo znalosti štruktúry GUI

Z hľadiska vykonávania testovacích prípadov sa jedná o automatickú metódu testovania GUI. Využíva sa fakt, že operačné systémy, alebo webové prehliadače poskytujú rozhranie na simulovanie rôznych používateľských akcií a dovoľujú nahliadnuť do štruktúry ovládacích prvkov aktuálne spustenej aplikácie (napr. vyrendrovaný strom webovej aplikácie DOM). Nad takouto štruktúrou je možné vykonávať dotazy na ovládacie prvky (napr. pomocou jazyka XPath) a simulovať nad nimi konkrétne používateľské akcie.

Typický predstaviteľ tohoto prístupu je projekt *Selenium*[4], ktorý slúži na automatizáciu webových prehliadačov pomocou vytvoreného *API*[5]. Toto *API* implementujú vývojári internetových prehliadačov v externom programe, ktorý dokáže daný internetový prehliadač ovládať. Používateľ pomocou implementovaných volaní dokáže vykonávať dotazy na tieto externé programy a tak ovládať prehliadač. Za zmienku stoja aj rozhrania *pyatspi* pre operačný systém Linux a *win32api* pre operačný systém Windows, ktoré umožňujú pracovať s daným rozhraním operáčného systému. Pre automatizáciu GUI nad *pyatspi* je možné použiť knižnicu *dogtail*[6]. Pre automatizáciu Windows rozhrania je možné použiť *WinAppDriver*[7], ktorý pracuje nad podobným protokolom ako nástroj *Selenium*.

Sada nástrojov Selenium

Prvým projektom, ktorý by sa dal nazvať nástrojom pre automatizáciu prehliadača bol vytvorený už v roku 2004 [5] a nad jeho základom neskôr vznikla architektúra *Selenium Remote Control*, ktorá sa označuje ako Selenium 1. Nevýhodou tejto architektúry bolo priame injektovanie časti kódu v jazyku JavaScript pri načítaní webovej stránky, ktoré

[4] https://www.seleniumhq.org/
[5] https://w3c.github.io/webdriver/
[6] https://gitlab.com/dogtail/dogtail
[7] https://github.com/Microsoft/WinAppDriver

sa niekedy nezaobišlo bez problémov. [4] Tento kód slúžil na vykonávanie používateľských príkazov prijatých od *Remote Control Server*, ktorý slúžil ako prostredník medzi danou webovou aplikáciou a klientskou knižnicou.

Selenium 2 vniklo predstavením WebDriver API, ktoré implementovali ovládače (WebDrivers) vtedy najpoužívanejších prehliadačov a ďalších mobilných platforiem – Android a iPhone [2]. Vďaka tomuto ovládaču už naďalej nebola potreba injektovať kód kódu priamo do webovej aplikácie a okrem toho bolo možné Selenium využívať aj na automatizáciu mobilných aplikácií. Pre vývojárov daných platforiem zostáva nutnosť implementovať daný protokol, aby bolo možné danú platformu aspoň do určitej miery automatizovane testovať.

Aktuálne existuje verzia systému Selenium 3 (verzia 3.14.0), ktorá odstraňuje podporu pre Selenium RC (ktorá vo verzii Selenium 2 ešte bola) a zároveň táto verzia vznikla tesne po tom, ako konzorcium W3C zverejnilo dokument s protokolom WebDriver[8] ako odporúčanie pre ovládače webových prehliadačov [3]. Aktuálne framework Selenium sa skladá z nasledujúcich častí:

- Klientské knižnice, ktoré využíva vývojár alebo tester, pomocou ktorých je možné využívať WebDriver protokol s použitím konkrétneho ovládača WebDriver.Tieto knižnice sú dostupné pre bežné programovacie jazyky ako – Java, Python alebo C#.
- Ovládače *WebDriver*[9] pre platformy, o ktoré sa starajú vývojári jednotlivých platforiem a slúžia ako prostriedok na vykonávanie špecifikovaných príkazov pre danú platformu.
- Server *Selenium Standalone Server*, ktorý sa stará o vzdialené spúšťanie jednotlivých zaregistrovaných ovládačov *WebDriver* pomocou neho je možné automatizovane ovládať aplikácie mimo hostiteľského počítača. Tento server je možné používať v spojení so službou Docker[10], pomocou ktorej je možné jednoducho škálovať použité ovládače GUI.
- Nástroj *Selenium IDE*[11], ktorý funguje ako rozšírenie pre webové prehliadače, pomocou ktorého dokáže užívateľ vytvárať testovacie prípady pomocou metódy zaznamenaj a prehraj, ktorá je popísaná v nasledujúcej časti.

Klientské knižnice implementujú dve rozhrania – *WebDriver* a *WebElement*. Prvé z týchto rozhraní slúži na ovládanie jednotlivých ovládačov GUI. Pomocou neho je možné spustiť testovanú aplikáciu, meniť veľkosť okna, v prípade webových prehliadačov, je možné ovládať navigáciu prehliadača.

Ovládacie prvky GUI aplikácií sú reprezentované pomocou acyklického stromu, v prípade webových aplikácií sa táto reprezentácia volá *DOM*. Webové prehliadače poskytujú prostriedok na dotazovanie sa nad týmto stromom a túto skutočnosť využívajú aj jednotlivé ovládače *WebDriver*. Reprezentáciu *DOM* je možné využívať ako XML dokument a dotazovanie pomocou ovládačov je možné na základe niektorých špeciálnych atribútov jednotlivých prvkov tohoto dokumentu (napr. *Id* alebo *ClassName*). Okrem toho je možné využívať aj jazyk *XPath*[12] na dotazovanie sa vrámci celého dokumentu. Rozhranie *WebDriver* dokáže vrátiť prvý prvok alebo všetky prvky na základe poskytnutého selektoru (funkcie

[8] https://www.w3.org/TR/webdriver/
[9] https://www.seleniumhq.org/download/#thirdPartyDrivers
[10] https://github.com/SeleniumHQ/docker-selenium
[11] https://www.seleniumhq.org/selenium-ide/
[12] XML Path Language

find_element a find_elements). Tieto prvky implementujú rozhranie *WebElement*, pomocou ktorého je možné nad nimi simulovať používateľské udalosti, napr. click(), send_keys() a dotazovať sa na vlastnosti daného ovládacie prvku, napr. výška, šírka, aktuálna pozícia na stránke, apod. Nad ovládacími prvkami *WebElement* je možné sa znova dotázať na ďalšie vnorené prvky tohoto elementu.

2.2.3 Testovanie pomocou rozpoznávania objektov GUI

Rovnako ako predchádzajúci prístup, medzi automatické metódy sa radí aj testovanie pomocou rozpoznávania objektov GUI. Pri tomto druhu testovania sa jednotlivé používateľské akcie vykonávajú programovo prostredníctvom rozhrania, ktoré poskytuje GUI systém, ale s tým rozdielom, že nie je potrebné nahliadnuť do štruktúry GUI. Podľa vopred poskytnutej obrazovej informácie (napr. obrázok ovládacieho prvku) sa vyhľadá ovládací prvok na aktuálnej obrazovke a nad ním sa sa vykoná akcia.

Príkladom takéhoto prístupu je projekt *Sikuli*[13], ktorý využíva knižnicu *OpenCV*[14] pre vyhľadávanie objektov v obraze a knižnicu *Tesseract*[15] na rozpoznávanie textu, implementujúcu metódu *OCR* (angl. Optical Character Recognition).

Výhodou testovania pomocou rozpoznávania objektov je, že sa testuje to, čo by používateľ naozaj videl. Nevýhodou je samotný proces vyhľadávania objektov v obraze, pretože je časovo náročný a výsledky vyhľadávania nie sú vždy presné.

2.2.4 Metóda zaznamenaj a prehraj

Metóda zaznamenaj a prehraj sa snaží predchádzať zdĺhavému vytváraniu testovacích prípadov a využiť výhody automatického vykonávania testovacích prípadov z predchádzajúcich prístupov. Používateľ pomocou nástroja, ktorý podporuje takýto prístup (napr. už spomänuté Selenium IDE), dokáže nahrávať svoje udalosti v SUT, ktoré sú zaznamenávané ako jednotlivé kroky v testovacom prípade. Po dokončení nahrávania je pomocou príslušného nástroja možné tieto udalosti automatizovane prehrať.

Tento prístup pre vytváranie testovacích skriptov je veľmi jednoduchý a nevyžaduje pokročilé princípy programovania od používateľa. Nevýhodou je, že pri každej malej zmene UI je nutné znova nahrávať testovacie prípady, ktoré by boli ovplyvnené touto zmenou.

[13] http://sikulix.com/
[14] https://opencv.org/
[15] https://github.com/tesseract-ocr/tesseract

Kapitola 3

Testovanie na základe modelu

Testovanie na základe modelu systému (MBT) využíva behaviorálny model testovaného softvéru. Testovacie prípady sú automaticky vygenerované na základe daných kritérií. Testovacie prípady je možné spúšťať nad testovaným softvérom [21]. V tejto kapitole je bližšie predstavený tento proces, jeho prínosy a obmedzenia pri testovaní. V ďalšej časti kapitoly sú vybrané a detailne popísané existujúce nástroje, ktoré využívajú princípy MBT a je možné ich využiť aj pri testovaní GUI. Pri každom nástroji sú zhrnuté výhody a nevýhody.

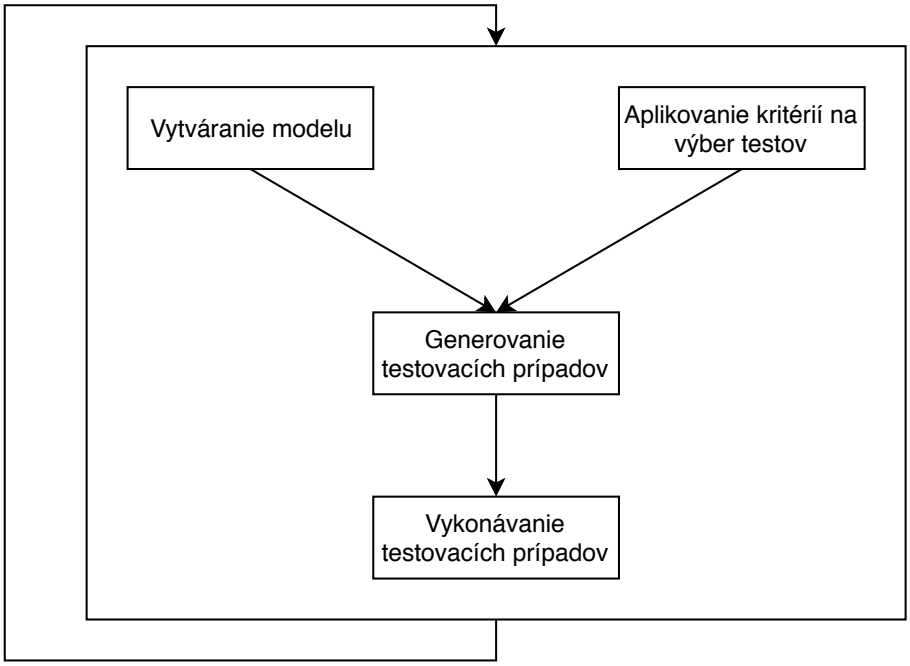
3.1 Proces testovania na základe modelu

Testovanie softvéru nie je len o vytváraní testovacích prípadov a ich spúšťaní. Pri návrhu softvéru sa väčšinou vytvára testovací plán pre jednotlivé časti aplikácie a na základe tohto plánu sú implementované testovacie prípady [19]. Takýto proces je väčšinou manuálny a niekedy aj pracný, ale existujú metódy, ktoré sa ho snažia automatizovať. Jednou z nich je práve testovanie na základe modelu [11]. Obrázok 3.1 zachytáva jednotlivé fázy pri MBT.

Všeobecný priebeh pri testovaní na základe modelu môžeme rozdeliť do štyroch hlavných aktivít [11]:

1. **Vytváranie modelu**. Model testovaného systému vychádza z požiadaviek na tento systém, popr. z existujúcich dokumentov so špecifikáciou softvéru. Tento model slúži ako abstrakcia pre testovacie požiadavky pre testovaný systém [21]. Pre testovacie modely sa niekedy využívajú aj časti z návrhu testovaného systému, ale nie je vhodné používať celé návrhy, pretože potencionálne chyby v návrhu by sa mohli odzrkadliť aj v testovanom modeli [18]. Model musí byť presný a kompletný, aby bolo možné automaticky vygenerovať testy a popr. aj testovacie *orákulum*. Pri modelovaní je dôležité nájsť vhodnú úroveň abstrakcie, aby bol vytvorený model jednoduchší ako testovaný softvér. V opačnom prípade by bola validácia testovaného modelu rovnako náročná ako validácia testovacieho softvéru [11].

2. Testovací model je len abstrakciou testovaného softvéru, ale väčšinou je tento model natoľko komplexný, že je možné z neho vygenerovať nekonečne mnoho testovacích prípadov (napr. v prípade, že máme model reprezentovaný konečným automatom a existuje v ňom aspoň jeden cyklus). Na výber vhodných testovacích prípadov alebo na splnenie určitého kritéria pokrytia, musí tester **aplikovať kritéria na výber testov** [11].



Obrázek 3.1: Diagram zachytávajúci pracovný postup pri testovaní na základe modelu [11]. Postup sa skladá zo štyroch hlavných častí. Je vidieť, že sa jedná o iteratívny proces, kedy každý krok postupu môžeme vyhodnotiť a na základe spätnej väzby môžeme upraviť model, upraviť kritéria na generovanie testov alebo vykonať opravy v testovanom softvéri, na základe vyhodnotených chýb. Pri úprave softvéru je potrebné počítať s úpravou, prípadne znovu vytvorením testovacieho modelu.

Kritériá na výber testov sa môžu vzťahovať k funkcionalite systému (kritéria výberu na základe požiadavkov), k štruktúre testovacieho modelu (pokrytie stavov, prechodov) alebo k pokrytiu dát [21].

3. Na základe testovacieho modelu a kritérií na výber testov sa automaticky **vygenerujú testovacie prípady** s cieľom vyhovieť všetkým kritériám. V závislosti na nástroji, je vygenerovaný testovací prípad s testovacími dátami a prípadne aj spustiteľným testovacím skriptom. Testovacie prípady sú typicky sekvencia akcií so vstupnými parametrami a očakávanými výstupnými hodnotami. Generátor sa môže snažiť minimalizovať generovanú testovaciu sadu, aby bola výsledná sada čo najmenšia, ale zároveň pokrývala všetky kritériá [11].

4. Posledným krokom je **spúšťanie testovacích prípadov**, ktoré sa nelíši od konvenčného testovania softvéru. Vygenerované testovacie prípady môžu byť vykonávané manuálne alebo automatizovane pomocou testovacieho prostredia. V druhom prípade je nutné, aby v predchádzajúcom kroku bol vygenerovaný aj testovací skript na spustenie testovacích prípadov, prípadne aby tester mal nástroj, adaptér, ktorý by dokázal vygenerované testovacie prípady vykonať. [21].

Výhody a nevýhody testovania na základe modelu

MBT sa snaží pozitívne ovplyvniť efektívnosť pri testovaní. Pomáha pri organizovaní a komunikovaní medzi testermi a inými členmi vývojového týmu. Výhodou je aj automatické generovanie testov, ktoré môže podliehať automatizovanému návrhu testov [9]. Existujú štúdie, ktoré ukazujú aj nárast objavených chýb v prípade použitia MBT oproti konvenčnému testovaniu, pretože vygenerované testovacie prípady dokážu často pokryť väčšiu časť SUT [20].

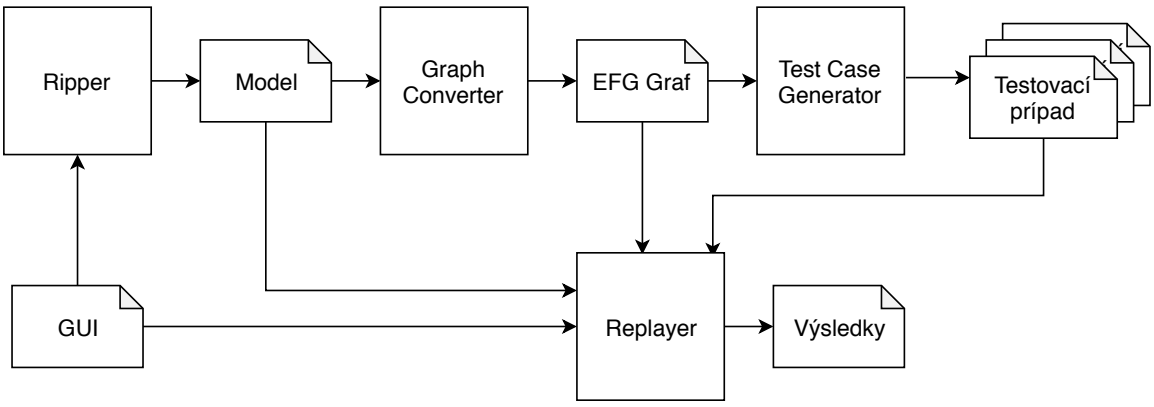
MBT sa od bežného testovania mierne líši. Zavedenie požívania tejto metódy môže znamenať finančné náklady na preškolenie testerov, prípadne na nákup softvéru podporujúceho MBT. Nevýhodou je aj analýza chýb, ktoré sa môžu vzťahovať ku chybnému modelu. Proces vývoja softvéru nie je vždy optimálny a tvorba modelu nemusí vždy vychádzať z aktuálnych požiadaviek [9]. Pri testovaní GUI vytváranie modelu väčšinou vychádza z implementovaného GUI, takže tento problém nie je v tomto prípade až taký významný.

3.2 Nástroje generujúce testovacie behy nad GUI

V tejto časti sú predstavené tri nástroje, ktoré využívajú princípy testovania na základe modelu. GUITAR a WebMate sú nástroje priamo určené na testovanie grafických používateľských rozhraní a nástroj MISTA okrem testovania GUI sa zameriava aj na ďalšie úrovne testovania. V poslednej časti kapitoly sú tieto nástroje zhodnotené.

3.2.1 GUITAR

Prvým z nich je sada nástrojov GUITAR[1], ktorý je jedným z najznámejších open-source nástrojov pre testovanie aplikácií s GUI na základe modelu [15]. Projekt je implementovaný v programovacom jazyku Java a pomocou zásuvných modulov je možné ho rozšíriť tak, aby bolo možné testovať aplikácie na rôznych operačných systémoch a popr. webové aplikácie. Na obrázku 3.2 je znázornený štandardný postup práce pri jeho používaní.



Obrázek 3.2: Postup práce pri používaní nástroja GUITAR [15]. Na obrázku sú znázornené jednotlivé nástroje z GUITAR, ich vstupy a výstupy.

Prehľad nástrojov obsiahnutých v sade GUITAR [12][15]:

- *Ripper* je nástroj pre automatické získanie modelu GUI z testovaného systému,

[1] https://sourceforge.net/projects/guitar/

- *Graph Converter* slúži na prevod modelu, ktorý je výstupom z nástroja *Ripper*, do grafu toku udalostí (angl. Event-Flow Graph), ktorý bude popísaný neskôr,
- *Test Case Generator* sluzi na vygenerovanie testovacích sád z vytvoreného grafu toku udalostí,
- *Replayer* je nástroj na automatické spúšťanie vygenerovaných testov z predchádzajúceho kroku.

Strom GUI a graf toku udalostí

GUITAR pre svoju činnosť využíva dve štruktúry prispôsobené na modelovanie GUI a generovanie testovacích prípadov. Prvou z nich je strom GUI, v skorších fázach vývoja GUITAR označovaný ako aj les GUI, ktorý slúži ako model GUI [12]. Je to graf, ktorý zachytáva štruktúru okien, čo sú uzly grafu, a hierarchické vzťahy medzi oknami sú zachytené hranami grafu. Okno, uzol grafu, je množinou ovládacích prvkov, ich vlastností a hodnôt a je reprezentované ako $S \subseteq W \times P \times V$, kde:

- W je množina ovládacích prvkov,
- P je množina vlastností, ktoré môžu ovládacie prvky z W nadobúdať (veľkosť, pozícia, farba textu, ...),
- V je množina hodnôt, ktoré môžu vlastnosti P nadobúdať.

Strom GUI je definovaný ako trojica $G = (\Psi, T, E)$, kde:

- Ψ je množina okien,
- $T \subseteq \Psi$ je množina hlavných okien,
- $E \subseteq \Psi \times \Psi$ je množina orientovaných hrán v tvare (x, y), kde udalosť vykonaná v okne x otvorí okno y.

Strom GUI zachytáva iba štruktúru GUI a nie je vhodný na generovanie testovacích prípadov, ale behom vytvárania tejto štruktúry sa získavajú aj ďalšie informácie o ovládacích prvkoch a typoch udalostí, ktoré je možné s týmito prvkami vykonávať. O každom okne sa zaznamenáva aj modálnosť okna. Na základe modálnosti okien sa definujú komponenty GUI ako dvojice (M, N), kde M je modálne okno a N je nemodálne okno a každé okno N je vyvolané udalosťou z okna M alebo N. Medzi jednotlivými komponentami je možné vytvoriť tzv. integračný strom $(Z, R, D$, kde Z je množina komponent, R je hlavná komponenta a D popisuje vzťah medzi komponentami [13].

Z týchto informácií je možné pre komponentu C vytvoriť graf toku udalostí (angl. *Event-Flow Graph*), ktorý zachytáva všetky možné udalosti, ktoré je možné vykonávať v jednotlivých oknách a udalosti, ktoré sú spojené s prechodom do iných okien [12] a ktorý je definovaný ako $EFG = (V, E, B, I)$, kde:

- V je množina udalostí v komponente C,
- $E \subseteq V \times V$ je množina orientovaných hrán, pre ktorú platí, že $\forall(e_i, e_j) \in E$, udalosť e_j môže nasledovať po vykonaní udalosti e_i,
- $B \subseteq V$ je množina udalostí, ktoré je možné vykonať po spustení komponenty C,
- $I \subseteq V$ je množina udalostí, ktoré vyvolávajú modálne okno.

Ripper

Nástroj *Ripper* pomocou automatickej interakcie s GUI, získava údaje o jeho oknách a ovládacích prvkoch. Z týchto údajov vytvára štruktúru stromu GUI popísaného vyššie 3.2.1 a následne ho vygeneruje na výstup vo formáte *XML*.

Pre správne fungovanie nástroja musí používateľ na vstupe zadať konfiguračný súbor obsahujúci [15]:

1. informácie pre spustenie SUT, z ktorého sa bude získavať model,
2. identifikáciu jednotlivých okien pomocou ich titulkov,
3. identifikáciu ovládacích prvkov, ktoré ukončujú aplikáciu,
4. špecifikovanie počiatočného okna,
5. špecifikovanie používateľských vstupov, pre prvky, ktoré vyžadujú vstupné dáta.

Graph Converter

Na prevod stromu GUI do grafu toku udalostí, popísaných v 3.2.1, sa využíva nástroj *Graph Converter*. Princíp fungovania je, že najprv sa identifikujú všetky možné počiatočné udalosti s ovládacími prvkami (napr. click-Ok, click-Edit, click-Cancel, ...). Pre každú takúto udalosť sa pomocou rekurzívneho algoritmu vypočíta množina možných následníkov [13].

Test Case Generator

Nástroj automaticky generuje testovacie prípady v tvare $\langle e_1, e_2, \ldots, e_n \rangle$ z grafu toku udalostí, ktorý používateľ predá na vstup. Používateľ musí špecifikovať aj algoritmus, ktorým sa má daný graf prechádzať a generovať cesty v grafe, popr. môže implementovať vlastný algoritmus ako zásuvný modul. Nástroj slúži aj ako generátor testovacích vstupov (napr. pre vygenerovanie vstupov textových polí sa využijú informácie s konfiguračného súboru pre *Ripper*). V prípade, že sa vygeneruje cesta v tvare (e_1, e_2) a udalosť e_1 nie je možné vykonať z počiatočného stavu, tak sa cesta vhodne doplní [15].

Používateľ dokáže vo vstupnom súbore špecifikovať kritéria pokrytia pre model GUI, z ktorých následne vychádzajú jednotlivé testovacie prípady. Kritéria pokrytia sa môžu týkať:

- pokrytia jednotlivých udalostí,
- pokrytia párov udalostí,
- pokrytia sekvencií udalostí dĺžky n.

Vymenované pokrytia predstavujú pokrytie všetkých uzlov grafu EFG, všetkých hrán EFG a ciest o dĺžke n v grafe EFG.

Replayer

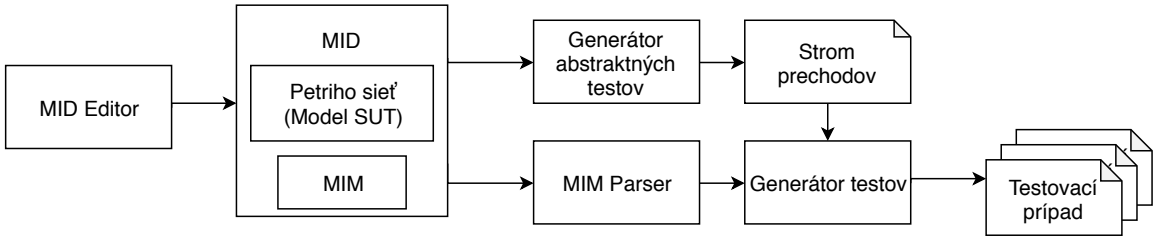
Pre vygenerované testovacie sady je pripravený nástroj *Replayer*, ktorý slúži na ich spúšťanie a ich vyhodnocovanie. Používateľ ma možnosť sledovania a nahrávania priebehu

testovania aplikácie. Naviac je možné vytvoriť zásuvný modul pre sledovanie iba konkrétnych aspektov aplikácie. Vytvorené nahrávky je možné použiť pre kontrolu ďalších spustení nástroja *Replayer* [15].

Replayer poskytuje dve testovacie *orákulá* na overenie či testovací prípad bol vykonaný v poriadku. Prvé z nich kontroluje vlastnosti ovládacích prvkov s modelom vytvoreným pomocou *Ripper* po vykonanej udalosti. Používateľ môže nastaviť, čo sa má kontrolovať a ako často sa má kontrola oproti modelu vykonať. Druhé *orákulum* kontroluje jednotlivé stavy aplikácie naprieč rôznymi testovacími prípadmi [15].

3.2.2 MISTA

Nástroj *MISTA*[2] (Model-based Integration and System Test Automation) sa zameriava na funkčné testovanie, testovanie bezpečnosti a regresné testovanie aplikácií, a teda nie je určený iba na testovanie grafických používateľských rozhraní. Je voľne dostupný a implementovaný v Jave. Pre modelovanie aplikácie využíva Petriho siete. Poskytuje prostriedok (*MIM- Model implementation mapping*) na mapovanie modelu na implementované konštrukcie v SUT. Na obrázku 3.3 sú znázornené jednotlivé časti nástroja MISTA [22].



Obrázek 3.3: Znázornený tok dát pre vygenerovanie testovacích prípadov a jednotlivé časti nástroja, ktoré vstupujú do tohoto procesu. Používateľ pomocou nástroja *MID Editor* podľa požiadaviek na testy pre SUT namodeluje Petriho sieť a pre jednotlivé časti Petriho siete vytvorí pomocou MIM mapovanie na SUT (cesta k spusteniu, mapovanie prechodov na jednotlivé metódy, ...). Generátor abstraktných testov najprv vytvorí strom dosiahnuteľných značení Petriho siete a na jeho základe vytvorí strom prechodov. Generátor testov prevedie strom prechodov vzhľadom k MIM a vygeneruje konkrétne testovacie prípady vo vybranom jazyku a testovacom framework [23].

Petriho siete a popis MIM

Používateľ pomocou nástroja, *MID Editor*, dokáže modelovať SUT pomocou Petriho siete [23], ktoré poskytujú rozsiahle analytické možnosti a dokážu modelovať systém viac konkrétnejšie [10]. Nástroj MISTA využíva mierne upravenú Petriho sieť s inhibičnými hranami. Pertriho sieť je definovaná ako $N = (P, T, F, I, L, \varphi, M_0)$, kde: [23]

- P je konečná množina miest,
- T je konečná množina prechodov,
- $F \subseteq (P \times T) \cup (T \times P)$ je prechodová relácia,
- $I \subseteq (P \times T)$ konečná množina inhibičných hrán,

[2] http://cs.boisestate.edu/~dxu/research/MBT.html

- $L \subseteq F \cup I$ je funkcia návestí,
- $\varphi \subseteq T \times A$ je strážna funkcia pre prechody $t \in T$, kde A je množina formúl predikátovej logiky prvého rádu,
- M_0 je množina počiatočných značení Petriho siete N.

Prechod Petriho siete N môže modelovať udalosť alebo funkciu v systéme. Môže byť obohatený o strážnu podmienku, ktorá špecifikuje, či je možné daný prechod vykonať, alebo nie. Hrany medzi miestami a prechodmi, môžu byť označené návestím, ktoré modeluje premenné systému [23]. Miesto Petriho siete N modeluje a udáva vstupnú, resp. výstupnú podmienku pre prechod $t \in T$, v závislosti či sa jedná o miesto zo vstupnej množiny, resp. z výstupnej množiny prechodu t [22]. Vstupnú podmienku pre prechod t je možné negovať inhibičnými hranami. Počiatočné značenie udáva nastavenie systému alebo vstupné dáta systému a jednotlivé ohodnotenia miest modelujú aktuálny stav systému [23].

Popis MIM prevádza jednotlivé časti vytvorenej Petriho siete (miesta, prechody, ...) na konkrétne komponenty modelovaného systému. Miesta a prechody je možné previesť na jednotlivé metódy (funkcie) testovaného systému. Ďalej je možné označiť miesta, ktoré slúžia na nastavenie systému. Je možné aj špecifikovať ďalšie časti kódu, ktoré slúžia ku generovaniu konkrétnych testovacích prípadov v konkrétnom programovacom jazyku, ako napr. potrebné balíčky pre jednotlivé súbory s testami, časti kódu na spustenie pred testovacími prípadmi, konštanty, ... [23].

Generovanie testovacích prípadov

V porovnaní z predchádzajúcou časťou, ktorá závisela najmä na používateľovi a z pohľadu modelovania bola manuálna, generovanie testovacích prípadov prebieha automaticky. Na základe modelu a mapovania, prebieha vytváranie testov v dvoch fázach.

V prvej fáze je vytvorený strom dosiahnuteľných značení, ktorého uzly udávajú všetky označenia, ktoré je možné dosiahnuť z počiatočného označenia. Na základe tohto stromu je možné generovať abstraktné testovacie prípady s určitým pokrytím, ktoré sa môžu týkať pokrytia uzlov, prechodov, určitej dĺžky, alebo určitého cieľa, ktorý je definovaný ako značenie Petriho siete [23].

Druhá fáza sa týka vygenerovania konkrétnych testovacích prípadov zo stromu prechodov vzhľadom k mapovaniu, ktoré špecifikuje používateľ [23]. Na výstupe je sada testov, ktorú je možné následne spúšťať v konkrétnom *xUnit* framework.

3.2.3 Webmate

Z narastajúcou popularitou webových stránok sa čoraz viac vyvíjané nástroje na testovanie začali orientovať iba na tento druh GUI (napr. [16] [14]). Jedným z nich je aj nástroj Webmate[3], ktorý funguje ako *cloudová služba*, ktorá automatizovane prehľadáva GUI webovej aplikácie a vytvára z nej model. Tento model je reprezentovaný konečným automatom, v ktorom jednotlivé stavy modelu reprezentujú stavy aplikácie a udalosti medzi stavmi aplikácie sú hranami modelu [6]. Stav aplikácie je identifikovaný funkcionalitou, ktorú daná stránka poskytuje, a teda, ovládacími prvkami, ktoré sú viditeľné a je možné s nimi interagovať. Dva stavy aplikácie sa teda líšia vtedy, ak poskytujú rozdielnu funkcionalitu [7].

[3] https://testfabrik.com/

Hrana modelu vzniká práve vtedy, keď sa aplikácia pomocou interakcie s nejakým ovládacím prvkom dostane do iného stavu. Webmate využíva pre na identifikáciu ovládacích prvkov a identifikáciu udalostí, ktoré je možné s nimi vykonať, nástroj Selenium[4].

V článku [6] sa uvádza, že Webmate dokáže generovať testy s pokrytím všetkých stavov, popr. stavov, ktoré si používateľ vyberie. Využíva Dijkstrov algoritmus na vyhľadávanie najkratšej cesty z počiatočného stavu do zadaného stavu konečného automatu.

3.2.4 Zhodnotenie nástrojov na testovanie GUI pomocou testovania na základe modelu

Pri vyhodnotení výhod a nevýhod jednotlivých nástrojov, je možné sa pozerať na rôzne aspekty. Výber týchto aspektov je zobrazený v tabuľke 3.1. Nástroje GUITAR a MISTA pochádzajú z akademického prostredia a sú voľne dostupné, naproti tomu nástroj Webmate je proprietárny. Záleží na požiadavkách, či je to nevýhoda alebo nie, pretože v prípade zakúpenia nástroja je navyše dostupná podpora a s najväčšou pravdepodobnosťou aj dokumentácia, čo v prípade nástrojov GUITAR a MISTA tomu tak nie je.

Nevýhodou nástroja MISTA, ktorá sa v tabuľke 3.1 neuvádza, je že model je nutné si vytvoriť manuálne, čo v prípade GUI môže byť náročné. V prípade nástroja GUITAR je zas nutné špecifikovať všetky stavy, ktoré sa v systéme nachádzajú. Zároveň model, ktorý využíva GUITAR nie je dobre prispôsobený pre moderné webové aplikácie.

Nástroj	**Open-source**	**Platformy**	**Model**	**CC**
GUITAR	Ano	Linux, Windows, Web	FSM, EFG	Pokrytie stavov a hrán udalostí
MISTA	Ano	Web	Petriho sieť	Pokrytie miest, prechodov, pokrytie značenia
Webmate	Nie	Web	FSM	Pokrytie stavov

Tabulka 3.1: Porovnanie nástrojov vo vybraných aspektoch. Stĺpec *Platformy* udáva cieľovú platformu SUT, pre ktorú je možné nástroj použiť. Stĺpec *CC* udáva aké kritéria pokrytia si môže používateľ zvoliť pre generovanie testovacích prípadov.

[4] https://www.seleniumhq.org/

Kapitola 4

Analýza požiadaviek a návrh nástroja na generovanie testovacích behov nad GUI

V tejto kapitole sú popísané dve etapy vývoja nástroja pre generovanie testovacích behov nad aplikáciami s GUI. V kapitole 4.1 sú špecifikované požiadavky, ktoré musí spĺňať vytvorený nástroj, ďalej je popísaný konceptuálny návrh systému, a proces ako by s týmto nástrojom mal tester pracovať. Konceptuálny návrh zobrazuje dve hlavné časti – nástroj na vytváranie modelu s GUI, tzv. monitor a nástroj generujúci testovacie behy nad špecifikovaným modelom splňujúce kritéria pokrytia. Tieto časti by mali byť schopné pracovať súčasne, ale nemal by byť problém s nimi pracovať aj samostatne. Ďalšie kapitoly formálne popisujú niektoré časti systému a venujú sa štrukturálnemu a behaviorálnemu návrhu jednotlivých častí systému.

4.1 Analýza požiadaviek pre systém na generovanie testovacích behov nad GUI

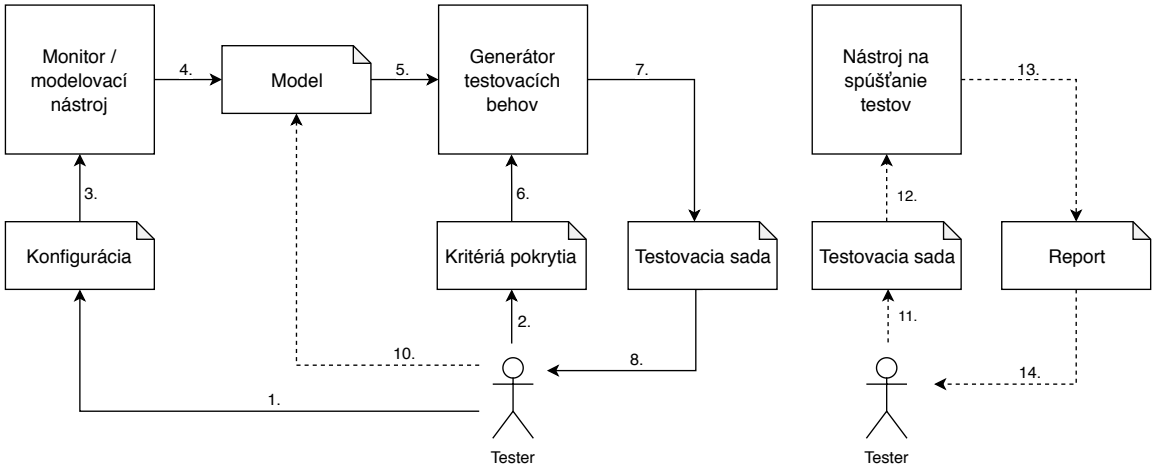
Id	Popis	Kategória	Závislosť
REQ.1	Nástroj musí generovať testovacie behy nad grafickým používateľským rozhraním SUT podľa poskytnutých kritérií pokrytia.	funkcionálny	-
REQ.2	Nástroj bude generovať testovacie prípady nad poskytnutým modelom GUI SUT.	funkcionálny	REQ.1
REQ.3	Testovacie prípady budú generované do pripraveného (meta)jazyku.	funkcionálny	REQ.1
REQ.4	Vygenerovanie testovacie prípady budú môcť byť spúšťané a vyhodnocované.	funkcionálny	REQ.1
REQ.5	Model GUI SUT bude reprezentovaný pomocou prechodového systému.	funkcionálny, model SUT	REQ.2
REQ.6	Existencia monitora na vytváranie modelu GUI SUT.	funkcionálny, model SUT	REQ.2
REQ.7	Model GUI SUT môže byť dodaný inou aplikáciou v špecifikovanom formáte.	funkcionálny, model SUT	REQ.2

Id	Popis	Kategória	Závislosť
REQ.8	Model GUI SUT zahŕňa možné stavy testovaného systému a prechody medzi nimi – používateľskými udalosťami.	funkcionálny, model SUT	REQ.2
REQ.9	Stav systému zahŕňa množinu ovládacích prvkov a množinu operácií, ktoré je možné nad ovládacími prvkami vykonať.	funkcionálny, model SUT	REQ.8
REQ.10	Prechodom medzi dvoma stavmi môže byť vykonaná používateľská udalosť s používateľským prvkom.	funkcionálny, model SUT	REQ.8
REQ.11	Používateľ bude mať možnosť zadávať kritériá pokrytia pomocou vytvoreného prostriedku (napr. jazyka pre špecifikáciu kritéria).	funkcionálny, CC	REQ.1
REQ.12	Kritéria pokrytia sa môžu týkať modelu systému – pokrytie uzlov, pokrytie hrán, pokrytie cesty špecifikovanej dĺžky.	funkcionálny, CC	REQ.11
REQ.13	Kritéria pokrytia sa môžu týkať softvérových artefaktov SUT – pokrytie všetkých widgetov aspoň raz, pokrytie vybraných widgetov aspoň raz, pokrytie vybraných widgetov n-krát.	funkcionálny, CC	REQ.11
REQ.14	Nástroj bude mať možnosť komunikovať s aplikáciami pre získanie rôznych kombinácii vstupných dát *T-Wise Coverage* alebo *Base Choice Coverage*, ktoré spadajú do testovania na základe domén.	funkcionálny, CC	REQ.11
REQ.15	Monitor bude na základe zachytených používateľských akcií a zachytených zmien GUI vytvárať model GUI.	funkcionálny, model SUT	REQ.6
REQ.16	Nástroj bude schopný vygenerovať testovacie behy v priebehu vytvárania modelu.	funkcionálny, model SUT	REQ.1
REQ.17	SUT môže byť aplikácia spĺňujúca *WIMP* alebo webová aplikácia, nemala by to byť 3D aplikácia, počítačová hra, alebo iný real-time systém.	nefunkc., SUT	REQ.15
REQ.18	Nástroj by mal byť schopný generovať testovacie behy pre SUT pre bežné OS – Windows, Linux.	nefunkc., SUT	REQ.15
REQ.19	Vytvorený systém by mal byť dostupný na bežných desktopových strojoch.	nefunkc.,	REQ.15

4.2 Konceptuálny návrh systému

Na základe požiadaviek bol vytvorený konceptuálny návrh 4.1 systému, ktorý sa skladá z niekoľkých častí, pomocou ktorých užívateľ – tester, by mal dokázať vygenerovať testovacie behy pre SUT. Prvou z nich je nástroj na vytváranie modelu GUI SUT, ktorý spúšťa SUT s cieľom automaticky získať model GUI postupným simulovaním používateľských udalostí nad GUI a zaznamenávaním získaných informácií do modelu v bližšie špecifikovanom formáte. Tento nástroj je nazvaný ako Monitor GUI, pretože sleduje (monitoruje) zmeny GUI, ktoré sú spôsobené používateľskými udalosťami. Vytvorený model z Monitoru môže

užívateľ použiť na vygenerovanie testovacích behov, ktoré spĺňajú užívateľom špecifikované kritéria pokrytia. Tieto behy môže užívateľ vhodne formátovať, aby sa dali ďalej použiť ako testovacia sada pre iný externý nástroj, napr. jeden z *xUnit rámcov*, ktorý poskytuje prostriedky pre spúšťanie testovacích behov, reportovanie chýb, apod.



Obrázek 4.1: Konceptuálny návrh nástroja na generovanie testovacích behov nad GUI. Popisuje tri základné podsystémy – modelovací nástroj, generátor testovacích behov a nástroj na spúšťanie testov. Užívateľ (tester) dodáva potrebné konfiguračné informácie pre monitor (1, 3), ktoré obsahujú informácie pre pripojenie na SUT. Užívateľ (tester) dodáva aj kritéria pokrytia pre generátor testovacích behov (2,6) na dodaný model. Hrana 10. znázorňuje možnosť použitia generátora bez závislosti na monitor GUI. Nástroj na spúšťanie testov nie je súčasťou diplomovej práce, ale znázorňuje možnosť použitia vygenerovanej testovacej sady s nejakým existujúcim *xUnit framework*[2].

4.3 Formálna definícia modelu pre GUI

Základom pre model GUI bol zvolený prechodový systém $(S, \rightarrow)$, kde S je množina stavov systému GUI a $\rightarrow \subseteq S \times S$ je množina prechodov určujúca vzťah medzi jednotlivými stavmi. Základnou jednotkou pre modelovanie GUI je ovládací prvok. S ovládacími prvkami je možné vykonávať používateľské akcie, ako napr. stlačenie tlačidla myši nad tlačidlom alebo stláčanie kláves nad formulárovými prvkami. Vykonanie udalosti nad ovládacím prvkom môže byť v modeli znázornené ako prechod zo stavu, v ktorom sa aplikácia nachádzala, do nového stavu aplikácie, ktorý bol spôsobený vykonanou akciou. Teda používateľská udalosť popisuje vzťah medzi dvoma stavmi systému. V prípade, že rozšírime prechodový systém na tzv. označený prechodový systém $(S, \wedge, \rightarrow)$, tak definícia stavov zostáva nezmenená, ale definícia množiny $\rightarrow$ sa zmení na $\rightarrow \subseteq S \times \wedge \times S$, kde $\wedge$ je množina označení prechodov medzi stavmi, v tomto prípade je to množina možných používateľských akcií nad ovládacími prvkami. Množina $\wedge$ je teda popísaná ako $\wedge \subseteq W \times A$, kde W je množina ovládacích prvkov a A je množina používateľských akcií.

[0] http://xunitpatterns.com
[2] http://xunitpatterns.com

Stav aplikácie GUI je popísaný množinou ovládacích prvkov, ktoré daný stav obsahuje a ovládací prvok GUI je možné popísať jeho vlastnosťami, ako napr. typ ovládacieho prvku, jeho rozmery, pozícia, atp. Ovládací prvok $w = (L)$ je definovaný pre model ako:

- $L : K \to V$ funkcia návestí, kde K je množina návestí (vlastností) pomocou ktorých môže byť ovládací prvok popísaný a V je množina hodnôt, ktoré môžu tieto návestia nadobúdať.

Model grafického používateľského rozhrania pre potrebu MBT, je popísaný označeným prechodovým systémom, ktorý je rozšírený o ďalšie prvky potrebné informácie pre popis GUI. Model GUI je sedmica $G = (W, S, \Sigma, A, \alpha, \delta, init, L)$, kde:

- W je konečná množina ovládacích prvkov,
- S je konečná množina stavov, kde $\forall s \in S : s \in 2^W$,
- Σ je konečná vstupná abeceda,
- A je množina akcií, ktoré je možné vykonávať v danom systéme GUI, napr. $A = \Sigma^* \cup \{click, hover, select\}$, kde množina Σ definuje vstupy pomocou klávesnice,
- $\alpha \subseteq W \times 2^A$ je funkcia, ktorá pre ovládacie prvky $w \in W$ definuje možné akcie,
- $\delta \subseteq S \times W \times A \times S$ je prechodová funkcia, ktorá je definovaná ako $s \xrightarrow{w,a} s'$, kde $s, s' \in S \wedge w \in W \wedge a \in A \wedge w \in s \wedge a \in \alpha(w)$,
- $init \subseteq S$ je množina počiatočných stavov.

Definovaný model $G = (W, S, \Sigma, A, \alpha, \delta, init, L)$ využívajú obidva navrhované nástroje. Monitor GUI, ktorý z testovaného systému vytvára takto popísaný model a generátor behov, ktorý z vytvoreného modelu generuje testovacie behy pre testovaný systém.

4.4 Vytváranie modelu GUI

Táto kapitola sa bližšie venuje nástroju, ktorý bol ukázaný v konceptuálnom návrhu 4.1 ako Monitor GUI, ktorý by mal slúžiť na získavanie modelu z GUI testovanej aplikácie bez väčších zásahov používateľa. V algoritme 4.1 je popísaný navrhnutý proces, ktorý je použitý ako základ pre získavanie modelu z GUI. Vstupom tohoto algoritmu je GUI, ktoré je zabezpečené pomocou integrácie už s existujúcimi nástrojmi pre automatizáciu GUI, nazývané aj ovládač GUI (angl. *Driver*), napr. framework Selenium pre webové aplikácie. Výstupom algoritmu je model popísaný v predchádzajúcej kapitole 4.3.

Všetky potrebné vstupy, ktoré užívateľ musí špecifikovať a výstupy nástroja sú popísané v kapitole 4.4.1, v ktorej je popísaný aj jazyk, pomocou ktorého užívateľ môže upravovať beh Monitora. Jednotlivé moduly, z ktorých sa nástroj skladá, sú popísané v kapitole 4.4.2. Komunikácia medzi jednotlivými modulmi a vrámci modulov je popísaná v kapitole 4.4.3.

V kapitole je popísaná aj integrácia s nástrojom *Combine* z platformy *Testos*. Tento nástroj slúži na generovanie dát alebo parametrov, ktoré vyhovujú pokrytiu všetkých n-tíc, v testovaní na základe domén. Použitie tohoto nástroja je v Monitore použité na generovanie vstupných dát pre formulárové prvky.

Vstup : Grafické používateľské rozhranie
Výstup : Model GUI $G = (W, S, \Sigma, A, \alpha, \delta, init)$

Vytvor iniciálny model $G_0 = (W_0, S_0, \Sigma, A, \alpha_0, \delta_0, init)$, kde $S_0 = \{init\}$;
$i = 1$;
$actual\_state = init$;
while $G_{i-1} \neq G_i$ **do**
    Vyber ovládací prvok $w \in actual\_state$ a akciu $a \in \alpha(w)$;
    Vykonaj akciu a nad ovládacím prvkom w;
    Vytvor množinu W_{new} z aktuálnych ovládacích prvkov GUI;
    Vytvor odpovedajúci stav $s_{new} = \{w | \forall w \in W_{new}\}$;
    if $s_{new} \in S_{i-1}$ **then**
        $G_i = (W_{i-1}, S_{i-1}, \Sigma, A, \alpha_{i-1}, \delta_{i-1} \cup \{(actual\_state, w, a, s_{new})\}, init)$;
    else
        $G_i = (W_{i-1} \cup W_{new}, S_{i-1} \cup S_{new}, \Sigma, A, \alpha, \delta_{i-1} \cup \{(actual\_state, w, a, s_{new})\}, init)$;
    end
    $actual\_state = s_{new}$;
    $i = i + 1$;
end
return G_i;

Algoritmus 4.1: Pseudo-algoritmus popisujúci získavanie modelu z GUI.

4.4.1 Vstupné a výstupné dáta Monitoru GUI

V tejto kapitole sú analyzované potrebné informácie, ktoré musí užívateľ špecifikovať pred samotným spustením Monitoru. Tiež je tu ukázaný navrhnutý formát výstupu Monitora, ktorý sa skladá z dvoch častí, a to zo získaného modelu GUI, ktorý bol špecifikovaný v kapitole 4.3 a zoznamom udalostí, ktoré viedli ku získaniu daného modelu GUI. Navrhnutý model vo formáte JSON je možné vidieť na obrázkoch 4.2 a 4.3. Formát súboru s vykonanými udalosťami je možné vidieť na obrázku 4.4.

Model aj zoznam udalosti sa dá znova použiť ako vstup pre Monitor, v takom prípade udalosti, ktoré musí Monitor vykonať sú načítané z daného súboru a model slúži ako zdroj referencií pre ovládacie prvky. Užívateľ môže upraviť súbor s udalosťami a takým spôsobom upraviť upraviť beh Monitora. Je možné špecifikovať inú sekvenciu udalostí, pre každý stav a je možné upravovať vstupné dáta pre používateľské udalosti, ktoré sa dajú parametrizovať. Zmena behu Monitora používateľom špecifikovanými vstupmi slúži ku aktualizácií stavov a udalostí vo vstupnom modeli GUI, najmä v prípade kedy samotný Monitor nedokáže uhádnuť vstupnú kombináciu udalostí a dát, napr. v prípadoch kedy možné ďalšie stavy aplikácie sa nachádzajú až po prihlásení od administračnej časti testovaného softvéru.

Užívateľ musí pre získavanie modelu GUI dodať nasledujúce konfiguračné informácie:

- názov ovládača GUI, ktorý sa bude používať pre interakciu s testovanou aplikáciou,
- konfiguračné informácie pre zvolený ovládač GUI, v ktorých musí byť aj informácia k spusteniu GUI,
- špecifikované ovládacie prvky, s ktorými bude Monitor GUI vykonávať používateľské akcie. Tieto ovládacie prvky sú definované špeciálnymi selektormi, pomocou ktorých ich dokáže ovládač GUI vyhľadať v GUI.

```
 ...
"initial_states": [
     "state_1"
],
"states": {
     "state_1": {
          "labels": {
               "widgets": [
                    {
                         ...
                         "name": "input_0",
                         ...
```

Obrázek 4.2: Časť modelu, ktorý je výstupom z Monitora GUI. Monitor pre každý nový nájdený stav generuje unikátny identifikátor, v tomto prípade je to identifikátor `state_1` a podobne aj pre každý ovládací prvok je vygenerovaný unikátny identifikátor, ktorý je uložený do návestia *name*.

```
 ...
"edges": {
     "edge_1": {
          "from": "state_1",
          "to": "state_1",
          "labels": {
               "name": "input_0",
               "event": "send_keys",
               "params": ["test"]
               ...
```

Obrázek 4.3: Časť modelu, ktorý je výstupom z Monitora GUI. Pre každú vykonanú používateľskú udalosť je generovaný unikátny identifikátor. Táto udalosť obsahuje referenciu na stav, v ktorom bola vykonaná a referenciu na stav, do ktorého daná hrana smeruje. Zároveň obsahuje referenciu na ovládací prvok, ktorý musí existovať v stave, v ktorom bola udalosť vykonaná.

V prípade, že užívateľ potrebuje spustiť Monitor znova s upraveným súborom udalostí, je to nutné špecifikovať príznakom *update_model* a pri tomto príznaku je nutné špecifikovať:

- model GUI, ktorý Monitor pri načítaní deserializuje a následne aktualizuje v prípade, že Monitor objaví pri prehľadávaní GUI nový stav,
- súbor so špecifikovanými používateľskými udalosťami, ktorý užívateľ môže ľubovoľne upravovať podľa daných pravidiel.

Jazyk pre popis používateľských udalostí

V tejto časti je definovaný jazyk pomocou, ktorého sú zapisované vykonané udalosti do súboru a zároveň pomocou ktorého môže užívateľ špecifikovať vstupné udalosti pre Monitor GUI. Tento jazyk vychádza z dostupných používateľských operácií bežne používaných ovládačov GUI. Jazyk je definovaný pomocou Backusovej–Naurovej formy na obrázku 4.5. Gra-

```
events:
     state_1:
       -
         - input_0.send_keys("Admin")
         - input_1.send_keys(${param1}, "@", ${param2})
         - button_0.click()
     state_2:
       - button_0.click()
input_data:
     param1:
       - param1 = "Admin"
       - param1 = "user"
     param2:
       - param2 = "server.com"
       - param2 = "server.co.uk"
```

Obrázek 4.4: Návrh súboru s vygenerovanými používateľskými udalosťami s úpravami od používateľa. Užívateľ môže do časti `"input_data"` špecifikovať parametre pre nástroj *Combine* typu `"string"`. Tieto parametre môže používateľ využívať v udalostiach typu `"send_keys"` a všetky používateľské akcie, ktoré obsahujú Combine parameter, musia byť syntakticky oddelené do zvláštneho bloku, a všetky udalosti z daného bloku budú vyhodnotené toľko krát, koľko bude prijatých vstupných hodnôt z nástroja Combine. Takýto blok ma za úlohu označovať vstupné operácie pre jeden formulár v GUI.

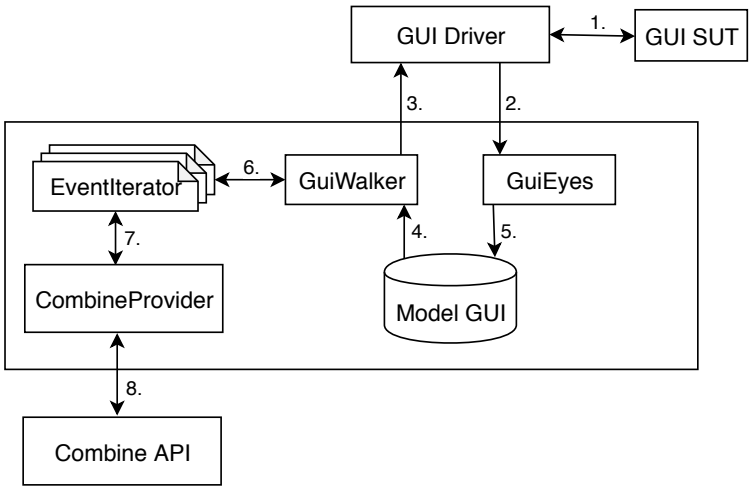
```
<command> ::= PROVIDER_NAME . <event_list>
<event_list> ::= <event>
          | <event_list> . <event>
<event> ::= click ( )
          | clear ( )
          | hover ( )
          | select ( STRING )
          | send_keys ( string_params )
<string_params> ::= <string_param>
          | <string_params> , <string_param>
          | send_keys ( string_params )
<string_param> ::= STRING
          | DATA_ID
```

Obrázek 4.5: Gramatika v Backusovej–Naurovej forme popisujúca možné používateľské udalosti a ich parametrizáciu.

matika používa terminál `PROVIDER_NAME`, ktorý označuje identifikátor ovládacieho prvku, ktorý sa môže skladať z malých a veľkých znakov, čísel a podčiarkovníku. Terminál `STRING` predstavuje v úvodzovkách ľubovoľný text a terminál `DATA_ID` predstavuje parameter, ktorý bude vyhodnotený nástrojom *Combine*, a tento terminál sa môže skladať z neprázdnej postupnosti malých, veľkých znakov, čísel a podtržníku a táto postupnosť musí začínať znakmi `${` a musí byť ukončená pomocou `}`.

4.4.2 Štrukturálny návrh Monitora

V tejto kapitole sú popísane jednotlivé navrhnuté moduly Monitora GUI a závislosti medzi nimi. Hlavný cyklus programu, ktorý je popísaný algoritmom 4.1, sa nachádza vo funkcii *gather_model()* v module *main*, ktorý využíva funkcionalitu modulov popísaný v nasledujúcich kapitolách a tieto moduly a vzťahy medzi nimi sú zobrazené na obrázku 4.6.



Obrázek 4.6: Diagram znázorňujúci jednotlivé moduly Monitora GUI. Ovládač GUI spolupracuje so systémom GUI (hrana 1.), ktorý využíva trieda *GuiEyes* na zisťovanie stavu systému a trieda *GuiWalker* na vykonávanie používateľských udalostí (2. a 3.), ktoré sú z modulu *monitor*. Tento modul zisťuje cestu medzi stavmi v modeli (4.) resp. zapisujú nové informácie (5.) – stavy a hrany modelu. Trieda *GuiWalker* využíva funkcionalitu iterátoru pre získavanie ďalších používateľských udalosti (6.), ktoré môžu obsahovať parametre, ktoré musí vyhodnotiť nástroj *Combine* (7. a 8.).

Modul pre spracovanie vstupu

V tomto module sú funkcie starajúce sa o spracovanie vstupu od používateľa. Nástroj môže mať viac rozhraní – rozhranie z príkazovej riadky alebo rozhranie REST API, kedy je nástroj použitý ako bežiaca webová služba. Modul obsahuje funkciu na spracovanie vstupov z príkazovej riadky *parse_cli_arguments()* a funkciu na spracovanie požiadaviek *parse_api_start_request*. Súčasťou tohto modulu je aj trieda na spracovanie súboru s používateľskými udalosťami *EventConfigurationParser* a jednotlivé udalosti v súbore sú spracované parserom v triede *EventParser*.

Modul reprezentujúci model GUI

Tento modul obsahuje model GUI navrhnutý v kapitole 4.3, a má za úlohu zapúzdriť všetky operácie, ktoré je možné s modelom a jeho jednotlivými časťami vykonávať. Hlavné časti modelu GUI sú popísané jednotlivými triedami nasledovne:

- ovládací prvok, sa nachádza v triede *Widget* a popisuje najdôležitejšie informácie o danom ovládacom prvku, okrem iného aj unikátny identifikátor, ktorý sa automaticky generuje a pomocou neho je ovládací prvok referencovaný naprieč celým systémom,

- stav systému GUI, je reprezentovaný triedou *State*, ktorá obsahuje referencie na všetky ovládacie prvky *Widget*, ktoré daný stav GUI obsahuje, a tiež pri vytvorení nového stavu je vygenerovaný unikátny identifikátor, ktorý reprezentuje daný stav,

- udalosť, je reprezentovaná triedou *Edge*, a obsahuje typ užívateľskej udalosti, referenciu na stav, v ktorom bola udalosť vykonaná, referenciu na stav do ktorého udalosť viedla a referenciu na ovládací prvok, nad ktorým bola udalosť vykonaná. Táto referencia je udávaná vygenerovaným identifikátorom daného ovládacieho prvku, teda nejedná sa o referenciu konkrétnej inštancie.

Všetky vyššie popísané triedy komponujú triedu *Model*, ktorá okrem jednotlivých referencií na vytvorené objekty obsahuje aj referencie na jednotlivé počiatočné stavy GUI. Okrem toho táto trieda obsahuje pomocnú metódu na serializáciu modelu do formátu JSON a statické metódy, ktoré slúžia na deserializáciu modelu zo súboru alebo z formátu JSON, ktoré vracajú novú inštanciu Modelu.

Modul pre integráciu s ovládačmi GUI

Táto časť systému je navrhovaná na splnenie požiadavku, aby nástroj dokázal generovať model pre testované systémy, ktoré môžu byť webové aplikácie, alebo desktopové aplikácie, pričom desktopové aplikácie môžu byť implementované pre rôzne operačné systémy. Pre každý takýto druh platformy existuje ovládač, ktorý zabezpečuje rozhranie pre automatizáciu daného prostredia a aplikácií, ktoré sa dajú v ňom spúšťať.

Modul pre integráciu sa skladá z troch časti. Prvou z nich je rozhranie *BaseDriver*, ktoré využíva monitor pre vykonávanie operácií s testovaným systémom. Aby bolo možné použiť konkrétny ovládač GUI v spojení s Monitorom, tak je nutné aby boli implementované metódy tohoto rozhrania a táto implementácia by sa starala o volanie príkazov konkrétneho ovládača. Ďalšou časťou je rozhranie *DriverBaseWidget*, ktoré je nutné implementovať zároveň pri implementácií rozhrania *BaseDriver* a určuje aké všetky udalosti je možné vykonať s ovládacími prvkami, pri použití daného ovládača GUI. Poslednou časťou je trieda *DriverFactory*, ktorá na základe názvu ovládača vracia konkrétnu inštanciu ovládača GUI, ktorý implementuje *BaseDriver*. Všetky názvy implementovaných ovládačov je nutné zaregistrovať v konfigurácií monitora.

Rozhranie *BaseDriver* obsahuje nasledovné metódy:

- pri spustení Monitoru sa využíva metóda *start()* a slúži pre získanie počiatočného stavu testovanej aplikácie pre model GUI,

- metóda *get_initial_states()* slúži na vrátenie všetkých definícií počiatočných stavov,

- metóda *close()* slúži na ukončenie testovanej aplikácie a následné ukončenie ovládača GUI,

- metóda *find_element()* slúži na vyhľadanie jedného ovládacieho prvku špecifikovaného pomocou selektoru,

- metóda *find_elements()* slúži na vyhľadanie všetkých ovládacích prvkov, ktoré odpovedajú špecifikovanému selektoru,

- pomocou metódy *get_events()* ovládač vracia všetky možné používateľské udalosti pre ovládacie prvky dané parametrom,

- pre vygenerovanie unikátneho selektoru pre jeden ovládací prvok slúži metóda *generate_selector()* a pomocou takéhoto selektoru je možné vyhľadať konkrétny ovládací prvok pri opakovanom navštívení stavu a
- metóda *take_screenshot()* slúži na zachytenie aktuálnej snímky obrazovky, ktorú je možné využiť na reprezentáciu stavov GUI a zároveň aj na reportovanie chyby pre užívateľa v prípade že nastane nejaký neočakávaný stav.

V závislosti od použitého ovládača GUI, selektor ovládacieho prvku pre metódy *find_element()* a *find_elements()* je väčšinou popísaný cestou vo formáte *Xpath*, ale selektor môže byť aj snímka obrazovky, ktorá zachytáva daný ovládací prvok. Užívateľ pri spustení Monitora musí špecifikovať ovládacie prvky, s ktorými môže nástroj interagovať a špecifikuje ich práve pomocou selektorov.

Modul events

Modul obsahuje triedy popisujúce jednotlivé používateľské udalosti a prácu s nimi. Je tu popis častí, ktoré spadajú do integrácie s nástrojom na generovanie kombinácie dát *Combine*. Trieda *PseudoEvent* reprezentuje používateľskú akciu, ktorá sa načítava so vstupného súboru s udalosťami. Táto trieda zapúzdruje identifikátor ovládacieho prvku z modelu GUI, názov používateľskej udalosti a zoznam parametrov pre používateľskú udalosť. Obsahuje aj metódu *transform()*, ktorej parametrom je referencia na ovládací prvok z použitého ovládača GUI a vracia objekt triedy *Event*, ktorá dedí priamo z *PseudoEvent*. *Event* sa odlišuje tým, že obsahuje priamu referenciu na ovládací prvok, nad ktorým sa má vykonať udalosť a obsahuje metódu *execute()*, ktorá volá danú používateľskú akciu nad ovládacím prvkom.

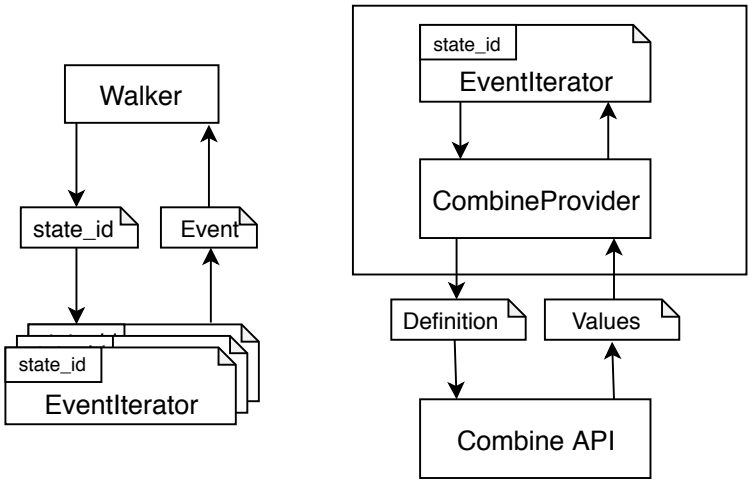
Používateľské akcie je možné parametrizovať, v závislosti od použitej udalosti. Užívateľ tieto parametre môže upravovať vo vstupnom súbore s udalosťami a okrem reťazcov pre formulárové prvky, užívateľ môže zadávať aj parametre spĺňujúce určité pokrytie z pohľadu testovania na základe domén. To je zabezpečené integráciou už s existujúcim nástrojom v platforme *Testos* nazvaným Combine[3]. Jednou z podmienok je použitie tejto integrácie iba v prípade potreby, tak aby nemusela existovať priama závislosť na tomto nástroji, teda aby nebol otvorený stály komunikačný kanál medzi nástrojom Monitor a Combine. Užívateľ môže špecifikovať špeciálne parametre vo vstupnom súbore s používateľskými udalosťami.

Pre tieto parametre je vytvorená trieda *CombineParameter* a trieda *CombineProvider*, ktorá sa stará o vyhodnocovanie konkrétnych parametrov. *CombineParameter* uchováva užívateľom špecifikované definície pre daný parameter a v prípade vyhodnotenia parametru, poskytuje vyhodnotené všetky hodnoty pre parameter. Nástroj *Combine* je dostupný ako samostatná služba a poskytuje REST API, ktoré na základe validného požiadavku poskytuje vyhodnotené kombinácie hodnôt pre zadaný požiadavok. Trieda *CombineProvider* sa práve stará o volanie služby *Combine*, na základe užívateľom špecifikovanej URL. Integrácia medzi jednotlivými modulmi a službou *Combine* je naznačená na obrázku 4.7.

Modul monitor

Tento modul sa stará o hlavnú funkcionalitu nástroja, ktorá je popísaná v algoritme 4.1 na riadkoch 5 až 12. Modul obsahuje triedu *GuiWalker*, ktorá sa stará o vykonávanie používateľských udalostí v jednotlivých stavoch aplikácie. Táto trieda si pre každý stav aplikácie drží informácie o možných udalostiach v danom stave a zároveň aj informácie o vykonaných

[3] https://combine.testos.org/



Obrázek 4.7: Diagram znázorňujúci integráciu modulu *events* s triedou *GuiWalker* z modulu monitor.

udalostiach. Pre každý stav je vedený iterátor *EventIterator* možných udalostí implementovaný v module *events*, ktorý pre aktuálny stav dokáže vrátiť nasledujúcu udalosť, ktorá sa má vykonať. Trieda *GuiWalker* obsahuje nasledujúce metódy:

- *init_events()* slúži na incializáciu iterátora pre špecifikovaný stav, ale iba v prípade, že pre daný stav iterátor neexistuje,
- pomocou metódy *do_step()* je vybraná nasledujúca používateľská udalosť pre stav daný parametrom a následne je vykonaná,
- metóda *walk()* berie ako argument zoznam udalostí a postupne jednu za druhou ich vykonáva
- metóda *find_path_to_unexplored_state()* je metóda na prehľadanie stavového priestoru, a slúži na vyhľadanie cesty do stavu, v ktorom ešte zostala nevykonaná používateľská udalosť. Výsledná cesta je zoznamom udalostí, ktoré sa majú postupne vykonať,

Druhou triedou, ktorá sa nachádza v tomto module je *GuiEyes*, ktorá ma za úlohu identifikovať jednotlivé stavy GUI. Obsahuje metódy *get_state()* a *compare_states()*. Prvá metóda predstavuje získanie aktuálneho stavu z GUI, čo je v algoritme 4.1 vidieť na riadku 8, ale pomocou tejto metódy sa získava aj počiatočný stav aplikácie. Táto metóda vytvára na základe získaných ovládacích prvkov novú inštanciu stavu *State* z modulu model. Druhá metóda slúži ako rozhodovací prostriedok na porovnanie aktuálneho stavu aplikácie s už existujúcimi stavmi v aktuálnom modeli. V algoritme 4.1 je tento proces znázornený na riadkoch 9 až 13.

Triedy *GuiWalker* a *GuiEyes* nemajú medzi sebou žiadnu referenciu, ale obidve využívajú ku svojej funkcionalite implementovaný ovládač GUI, ktorý slúži interakciu s GUI pre získavanie ovládacích prvkov pre stav systému GUI a zároveň sú pomocou ovládača aj vykonávane používateľské udalosti.

4.4.3 Behaviorálny popis Monitora

V predchádzajúcej časti boli popísané jednotlivé moduly nástroja Monitor a popísané triedy, ktoré sa starajú o funkcionalitu nástroja. V tejto časti je ukázaná komunikácia vrámci modulov a komunikácia medzi modulmi nástroja.

Činnosť nástroja je zahájená spracovaním parametrov od užívateľa a inicializovaním konfiguračných tried, ktoré sa starajú o poskytovanie konfigurácie pre jednotlivé moduly. V prípade, že sa jedná o spúšťanie Monitora so špecifikovaným vstupným modelom a zoznamom udalostí, je najprv volaná funkcia pre deserializáciu modelu GUI a následne je volaný parser na lexikálne a syntaktické spracovanie vstupných používateľských udalostí. Pri spracovávaní vstupných udalostí sa zároveň pripravujú inštancie triedy *EventIterator*, pre každý stav pre ktorý je vo vstupnom súbore špecifikovaná aspoň jedna udalosť. Spracované argumenty nástroja Monitor sú predané modulu *main*, kde sa nachádza hlavný cyklus programu, ktorý sa stará o prehľadávanie testovanej aplikácie.

Hlavný cyklus programu

Hlavný cyklus sa nachádza vo funkcii *gather_model()* a využíva metódy z modulu *monitor*. Vo tejto funkcii je na základe konfigurácie najprv inštanciovaný ovládač GUI pomocou triedy *DriverFactory* a tento ovládač je použitý pre vytvorenie inštancií tried *GuiWalker* a *GuiEyes*. Následne sa z týchto tried využívajú metódy pre získanie aktuálneho stavu GUI *get_state()*, vykonanie ďalšej používateľskej akcie *do_step()* a porovnanie aktuálneho stavu so stavom GUI po vykonaní používateľskej udalosti *comapre()*. V tomto cykle po vykonaní používateľskej udalosti sa pridáva nová hrana do modelu, a zároveň sa do modelu pridáva aj aktuálny stav, v prípade, že sa v modeli ešte nevyskytuje a zároveň je pre tento stav inicializovaný aj iterátor možných používateľských udalostí *EventIterator*, pomocou funkcie *init_events()*. Cyklus končí v prípade, že boli vykonané všetky možné udalosti zo všetkých stavov GUI.

Keď sú preskúmané všetky udalosti vo všetkých stavoch z vybraného počiatočného stavu, tak Monitor pokračuje načítaním ďalšieho počiatočného stavu a cyklus sa opakuje znova pre tento stav. Po ukončení cyklu pre všetky počiatočné stavy, je výsledný model a vykonané udalosti uložené do súboru, ktorý užívateľ špecifikuje pri spustení Monitora GUI. V prípade že to užívateľ nešpecifikuje názvy súborov pre výstup, tak je výstup zapísaný do súborov, ktoré sú špecifikované ako *model.out.json events.out.yaml*.

Vykonanie používateľskej udalosti

V momente keď sa zavolá metóda *do_step()* z triedy *GuiWalker*, tak na základe identifikátoru aktuálneho stavu je vybraný odpovedajúci *EventIterator* pre daný stav. Nad týmto iterátorom sa zavolá metóda *next()*, ktorá vracia nasledujúcu udalosť, ktorú je možné vykonať v danom stave GUI. V prípade, že pre daný stav už boli vykonané všetky používateľské akcie a zároveň už neexistuje stav, z ktorého by bolo možné vykonať ďalšiu používateľskú akciu, tak prehľadávanie GUI modelu končí. V opačnom prípade sú vzostupne zoradené stavy podľa počtu zostávajúcich udalostí, ktoré je potreba ešte preskúmať. Postupne pre každý stav z tejto postupnosti sa *GuiWalker* snaží nájsť cestu z aktuálneho stavu pomocou už zaznamenaných udalostí. V prípade, že takáto cesta existuje, tak pomocou metódy *walk()* je táto cesta vykonaná a následne je vykonaná udalosť, z nového aktuálneho stavu. Ak neexistuje žiadna cesta k nepreskúmanému stavu, tak Monitor znova načíta počiatočný stav aplikácie, pomocou ovládača GUI.

Vyhodnotenie používateľskej udalosti

Monitor GUI odlišuje dva druhy parametrov, ktoré, môžu prijať užívateľské akcie od používateľa. Prvým sú klasické reťazce a špeciálne klávesy reprezentované triedou *Event* a druhým druhom sú parametre pre vyhodnotenie nástrojom *Combine*. Udalosti, ktoré majú špecifikované aspoň jeden parameter určený pre nástroj *Combine*, musia byť zadávané v syntakticky odlíšených blokoch a v jednom bloku musia byť aspoň dva parametre s validnou definíciou, inak je pri sémantickom spracovávaní vyvolaná výnimka a program Monitor končí s chybou. Pre jednotlivé validné bloky sa vytvorí obálka *CombineProvider*, ktorá sa stará o jednotlivé udalosti v tomto bloku a jednotlivé parametre pre tieto udalosti. Inštancie udalostí *Event* a *CombineProvider* sú zaradené do zoznamu udalostí, z ktorého vracia iterátor udalosti keď sa nad ním zavolá metóda *next()*. Pred zavolaním *next()* sa iterátor pozerá na typ nasledujúceho objektu, a ak sa narazí na *CombineProvider*, tak sa pripraví požiadavok pre službu *Combine* so vstupnými parametrami a definíciami pre parametre. Treba podotknúť, že syntax definícií pre parametre sa nekontroluje syntakticky ani sémanticky a stará sa o to až služba *Combine* pri prijatí požiadavku. Ak sa pošle neplatná definícia pre parametre, tak je zaznamenaná chyba vrátená zo služby *Combine* a iterátor pokračuje so spracovaním ďalšej udalosti.

Služba *Combine* po zadaní validného požiadavku vracia zoznam o dĺžke l, ktorý obsahuje n-tice dát, pre zadaných n parametrov. Pre každý *CombineParameter* je vybraný zoznam odpovedajúcich hodnôt (tento zoznam má dĺžku l). Obálka *CombineProvider* obsahujúca daný blok používateľských akcií už s vyhodnotenými parametrami, ďalej vyhodnotí tento blok udalostí do zoznamu udalostí o dĺžke $l * pocet\_udalosti\_v\_bloku$, kde sa zoberie pôvodný blok udalostí a pre odpovedajúce udalosti sa dosadia hodnoty vyhodnotených parametrov a pre výsledný zoznam sa táto operácia zopakuje l-krát.

4.5 Generovanie testovacích behov

Druhá časť systému je nástroj generujúci testovacie behy pre poskytnutý model, nazvaný Generátor. V tejto kapitole je formálne popísaný testovací beh a kritéria pokrytia, ktorými je možné riadiť generovanie behov. Monitor GUI, popr. tester poskytuje pripravený model testovanej aplikácie a tester musí poskytnúť aj popísané kritéria pokrytia. Následne Generátor vytvorí množinu testovacích behov na svojom výstupe. Užívateľ môže na vstupe špecifikovať kritéria pokrytia pre model, ktoré sa môžu týkať jednotlivých častí modelu a môže dodať aj ďalšie testovacie požiadavky pomocou definovaného jazyka. Generátor spracuje kritéria pokrytia a na ich základe vygeneruje testovacie požiadavky pre model testovaného systému. Z vytvorených požiadaviek sa postupne vytvárajú testovacie behy, pomocou algoritmu 4.2.

4.5.1 Testovací beh a kritéria pokrytia

Neformálne je možné popísať testovací beh ako postupnosť príkazov, ktoré je možné vykonať v testovanom systéme. Pre predstavený model pre systémy s GUI v kapitole 4.3, ktorý vychádza z prechodového systému, sa testovacie behy skladajú postupností prechodov tohoto modelu. V tejto časti sú popísané aj prostriedky pre užívateľa, pomocou ktorých je možné špecifikovať kritéria pokrytia pre model GUI a zároveň pomocou pripraveného jazyka môže naviac určiť vlastné testovacie požiadavky, ktoré musia spĺňať generované testovacie behy.

Vstup : Množina testovacích požiadaviek TR
Výstup : Množina testovacích behov $Runs$

```
1  Runs = {};
2  Covered = {};
3  for tr ∈ TR do
4  |  if is_covered(tr, Runs) then
5  |  |  continue;
6  |  Run := tr;
7  |  for t ∈ TR, t ≠ tr do
8  |  |  TR := TR \ {t};
9  |  |  Covered := Covered ∪ {t};
10 |  |  Run := merge(Run, t);
11 |  end
12 |  if ¬ init(start(Run)) then
13 |  |  for i ∈ {start(t) |t ∈ Covered ∪ TR, is_init(start(t))} do
14 |  |  |  pref := find_path(i, start(Run));
15 |  |  |  if pref ≠ {} then
16 |  |  |  |  Run := merge(pref, Run);
17 |  |  |  |  break;
18 |  |  end
19 |  Runs := Runs ∪ {Run};
20 end
21 return Runs;
```

Alg. 4.2: Pseudo-algoritmus popisujúci generovanie testovacích behov z modelu a vybraného kritéria pokrytia.

Formálne je *testovací beh* π cesta zložená zo stavov, ovládacích prvkov a akcií s ovládacími prvkami a pre model $G = (W, S, \Sigma, A, \alpha, \delta, init)$ je definovaný ako:

$$\pi = \langle s_1 w_1 a_1 s_2 w_2 a_2 s_3 \dots s_{n-1} w_{n-1} a_{n-1} s_n \rangle \qquad \text{dĺžky } n > 1$$

kde pre jednotlivé zložky testovacieho behu π, takých že $s_1 \in init$ a pre $\forall i \in \{1, \dots, n-1\}$ platí:

$$s_i \in S, s_{i+1} \in S, w_i \in W, w_i \in s_i, a_i \in \alpha(w_i), s_{i+1} \in \delta(s_i, w_i, a_i)$$

Pre *testovací beh* $\pi = \langle s_1 w_1 a_1 s_2 w_2 a_2 s_3 \dots s_{n-1} w_{n-1} a_{n-1} s_n \rangle$ je možné definovať nasledujúce množiny:

- $\Pi(\pi) = \{s_1, \dots, s_n\}$, je množina navštívených stavov,
- $\Delta(\pi) = \{w_1 a_1, \dots, w_n a_n\}$, je množina vykonaných používateľských udalostí.

$$is\_init(s) = \begin{cases} true & s \in init \\ false & \text{otherwise} \end{cases}$$

Pre *testovací beh* $\pi = \langle s_1 w_1 a_1 s_2 w_2 a_2 s_3 \dots s_{n-1} w_{n-1} a_{n-1} s_n \rangle$ dĺžky n a model $G = (W, S, \Sigma, A, \alpha, \delta, init)$ a pre stav $s \in S$ je možné definovať nasledujúce funkcie:

$$start(\pi) = \begin{cases} s_1 & n \geq 2 \\ \emptyset & \text{otherwise} \end{cases}$$

$$end(\pi) = \begin{cases} s_n & n \geq 2 \\ \emptyset & \text{otherwise} \end{cases}$$

$$all\_subpaths(\pi) = \begin{cases} \{\langle s_i w_i a_i s_{i+1} \dots s_j \rangle \mid i \geq 1, j \leq n, i < j\} & n \geq 2 \\ \emptyset & \text{otherwise} \end{cases}$$

$$suffix(\pi, l) = \begin{cases} \langle s_{n-l+1} w_{n-l+1} a_{n-l+1} s_{n-l+2} \dots s_n \rangle & 2 \leq l \leq n \\ \langle\rangle & \text{otherwise} \end{cases}$$

$$prefix(\pi, l) = \begin{cases} \langle s_1 w_1 a_1 s_2 \dots s_l \rangle & 2 \leq l \leq n \\ \langle\rangle & \text{otherwise} \end{cases}$$

$$is\_covered(\pi, \chi) = \begin{cases} true & \pi \in all\_subpaths(\chi) \\ false & \text{otherwise} \end{cases}$$

$$find\_path(s, d) = \begin{cases} \langle s_1 w_1 a_1 s_2 \dots s_m \rangle & s_{i+1} \in \delta(s_i, w_i, a_i), 1 \leq i < m, s_1 = s, s_m = d \\ \langle\rangle & \text{otherwise} \end{cases}$$

Pre testovací beh $\pi = \langle s_1 w_1 a_1 s_2 w_2 a_2 s_3 \dots s_{n-1} w_{n-1} a_{n-1} s_n \rangle$ dĺžky n a testovací beh $\chi = \langle t_1 x_1 b_1 t_2 x_2 b_2 t_3 \dots t_{m-1} x_{m-1} b_{m-1} t_m \rangle$ dĺžky m vytvorené z modelu $G = (W, S, \Sigma, A, \alpha, \delta, init)$, je možné definovať nasledujúce funkcie:

$$concat(\pi, \chi) = \begin{cases} \langle s_1 w_1 a_1 \dots x_{m-1} b_{m-1} t_m \rangle & s_{n-1} = t_1, w_{n-1} = x_1, a_{n-1} = b_1, s_n = t_2 \\ \langle\rangle & \text{otherwise} \end{cases}$$

$$merge(\pi, \chi) = \begin{cases} concat(\pi, suffix(\chi, l)) & suffix(\pi, l) = prefix(\chi, l), l \leq min(n, m) \\ \emptyset & \text{otherwise} \end{cases}$$

Funkcia *find_path()* slúži na nájdenie najkratšej syntakticky dosiahnuteľnej cesty a funkcia *merge()* slúži na spojenie dvoch ciest, tak aby dohromady tvorili jeden syntakticky správny testovací beh.

Užívateľ má možnosť špecifikovať kritériá pokrytia pre model, ktorý je na vstupe pred generovaním testovacích behov. Kritéria sa vzťahujú k pokrytiu hrán grafu, a pre model GUI to sú jednotlivé prechody medzi stavmi. Je možné vygenerovať testovacie požiadavky pokrývajúce všetky hrany (*Edge Cogerage*, *EC*), alebo všetky dvojice hrán (*Edge Pair Cogerage*, *EPC*) alebo všetky hlavné cesty (*Prime Path Cogerage*, *PPC*) v modeli GUI.

V prípade, že užívateľ nešpecifikuje žiadne z daných pokrytí, tak môže definovať dĺžku testovacích požiadaviek a v takom prípade sa vygenerujú testovacie požiadavky, ktoré obsahujú postupnosť syntakticky dosiahnuteľných hrán o danej dĺžke. Vtedy jedna požiadavka môže obsahovať niekoľko krát jeden navštívený stav. Takto vygenerované požiadavky sa dajú obmedziť výhradne na jednoduché cesty, a teda jednotlivé požiadavky budú obsahovať iba unikátne stavy okrem ciest, ktoré budú začínať a končiť v rovnakom stave.

Jazyk pre definovanie dodatočných testovacích požiadaviek

Pri testovaní systémov s GUI, môže tester naraziť na potrebu pokrytia systému so špeciálnymi požiadavkami, ktoré by inak s vyššie popísanými kritériami pokrytia nikdy neboli obsiahnuté vo výslednej testovacej sade. Pre tento účel bolo navrhnutých päť operátorov, ktoré užívateľ môže využiť pre špecifikovanie testovacích požiadavkov pre behy v systéme a ich význam je nasledovný:

- unárny operátor `ITEM`, ktorým je možné špecifikovať jednu udalosť, ktorá sa má pokryť,
- binárny operátor predstavujúci zreťazenie dvoch operátorov `ITEM` za sebou,
- binárny operátor `+`, pomocou ktorého sa vygeneruje testovací požiadavok, ktorý začína udalosťou špecifikovanou na ľavej strane a končí udalosťou špecifikovanou na pravej strane. Tento testovací požiadavok bude obsahovať najkratšiu cestu medzi zadanými udalosťami a nastáva chyba, ak takáto cesta neexistuje,
- binárny operátor `*`, pomocou ktorého sa vygeneruje množina testovacích požiadavkov, ktoré obsahujú všetky jednoduché cesty, ktoré začínajú udalosťou špecifikovanou na ľavej strane a končia udalosťou špecifikovanou na pravej strane operátora,
- binárny operátor `*`, pomocou ktorého je možné násobiť cyklus, ktorý je špecifikovaný na ľavej strane operátora a je syntakticky odlíšený, tak aby sa dal rozoznať od predchádzajúceho operátora a tento cyklus je násobený celým číslom, väčším ako 0, ktoré je zadané na pravej strane operátora a nastáva chyba, ak je zadaný nevalidný cyklus, v ktorom posledná hrana sa nenachádza v predchodcoch prvej hrany.

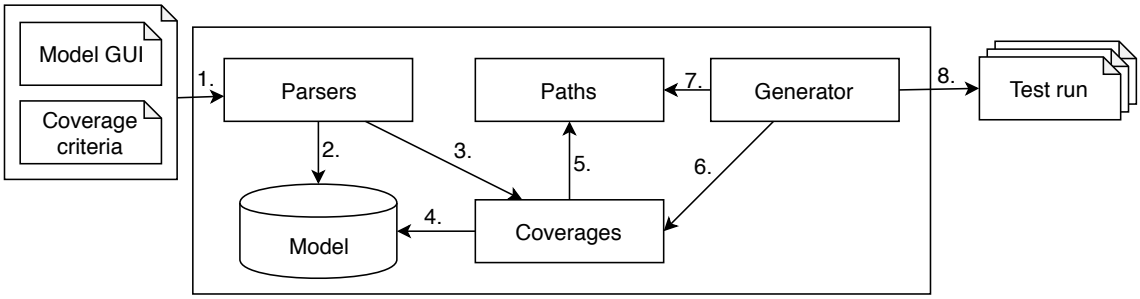
Gramatika týchto operátorov je popísaná v Backusej-Naurovej forme na obrázku 4.8.

```
<test_requirement> ::= <test_requirement> * <simple_item>
                     | <test_requirement> + <simple_item>
                     | <test_requirement> <simple_item>
                     | <simple_item>
<simple_item> ::= <cycle>
                     | <simple_item> ITEM_ID
                     | ITEM_ID
<cycle> ::= [ <test_requirement> ] * INTEGER
```

Obrázek 4.8: Gramatika v Backusovej–Naurovej forme popisujúca jazyk pre špecifikovanie testovacích požiadaviek. Token INTEGER predstavuje celé číslo väčšie ako 0 a token ITEM_ID predstavuje validný identifikátor hrany v modeli GUI.

4.5.2 Štrukturálny návrh Generátora

Táto kapitola popisuje modulárne rozčlenenie nástroja Generátor. Sú tu popísané hlavné moduly a ich štruktúra, a okrem nich nástroj obsahuje aj modul *parsers*, ktorý obsahuje funkcie na spracovanie argumentov od užívateľa a triedu *TestRequirementParser*, ktorá slúži na lexikálnu a syntaktickú analýzu definovaných testovacích požiadaviek používateľom. Jednotlivé moduly nástroja a vzťahy medzi nimi sú zobrazené na obrázku 4.9.



Obrázek 4.9: Diagram znázorňujúci jednotlivé moduly nástroja Generátor, jeho vstupy (hrana 1.) a výstupy (8.). Modul *parsers* deserializuje model (2.) a vytvorí konkrétnu inštanciu kritéria pokrytia (3.). Testovacie požiadavky vytvára modul *coverages* ako inštancie tried z modulu *paths* (5.), a na základe nich, modul *generator* vytvára testovacie behy (7.).

Modul model

Tento modul je podobný tomu, ktorý sa nachádza v predchádzajúcom nástroji Monitor, ale vyznačuje sa niektorými odlišnosťami potrebnými pre generovanie testovacích behov. Predstavený model GUI je príznačný tým, že môže obsahovať viac ako jeden prechod medzi dvomi stavmi modelu. Preto je modul rozšírený o triedu *AggregatedEdge*, ktorá agreguje jednotlivé hrany, ktoré majú spoločný počiatočný a koncový stav, pre každú agregáciu sa vytvorí nová inštancia tejto triedy. Formálne $\forall s, s' \in S$, kde S je množina stavov je vytvorená agregovaná hrana $AE = \{(s, w, a, s') \mid (s, w, a, s') \in \delta\}$, kde δ je množina prechodov modelu GUI. Vďaka tomuto je možné s modelom pracovať, ako keby medzi každou dvojicou stavu by bola maximálne jedna hrana. Jednotlivé agregované hrany sa vytvárajú pri deserializácii modelu.

V triede *Model* je implementovaná aj metóda na prehľadávanie stavového priestoru pre nájdenie najkratšej cesty medzi dvomi položkami modelu. Je to podobná metóda tej, ktorá sa nachádza v nástroji Monitoru, s tým rozdielom, že táto metóda je tu upravená tak, aby dokázala nájsť cestu medzi dvomi stavmi, ale aj cestu medzi dvomi hranami modelu. To umožní generátoru generovať cesty zložené z hrán, ale aj zo stavov.

Modul paths

Modul obsahuje triedy *TestRequirement*, ktorá reprezentuje testovací požiadavok a *TestRun*, ktorá reprezentuje testovací beh. Obidve tieto triedy dedia z triedy *Path*, ktorá implementuje operácie, ktoré je možné vykonávať s danými cestami:

- funkcia *init()* vracia pre danú cestu hodnotu typu `bool`, a vracia `true` podľa toho, či cesta začína v počiatočnom stave, ináč vracia `false`,
- funkcia *is_subsequence()* vracia `true`, ak inštancia danej cesty, je podcestou inej cesty typu *Path*, ktorá je predaná ako argument,
- funkcia *syntactic_sat()* vracia `true`, ak inštancia danej cesty, je syntakticky správne, teda každá hrana cesty, je predchodcom tej nadchádzajúcej hrany, s výnimkou poslednej.

Trieda *TestRequirement* obsahuje naviac metódu *is_covered()*, ktorá ako argument berie množinu ciest a vracia `true` iba v prípade, ak testovací požiadavok je podcestou aspoň jednej

cesty z danej množiny ciest. Testovací beh reprezentovaný triedou *TestRun* obsahuje metódu *merge()*, ktorá ako argument berie testovací požiadavok a tento požiadavok je pripojený na koniec testovacieho behu. Testovací beh má referencie aj na všetky testovacie požiadavky, ktoré daná beh pokrýva.

Modul s kritériami pokrytia

Modul s kritériami pokrytia obsahuje triedy *EdgeCoverage*, *EdgePairCoverage*, *PrimePathCoverage* a triedu *CustomCoverage*. Všetky spomenuté triedy implementujú rozhranie *BaseCoverage*, ktoré špecifikuje jednu metódu *generate()*, ktorá by mala slúžiť na vytvorenie testovacích požiadaviek na základe modelu, čo by mal byť vstupný parameter tejto metódy. Trieda *CustomCoverage* obsahuje algoritmus na vygenerovanie požiadaviek podľa užívateľom špecifikovaných parametrov – dĺžky požiadaviek a príznaku či sa má jednať o jednoduché cesty. V module sa nachádza aj trieda *CoverageFactory*, pomocou ktorej je možné vytvoriť konkrétnu inštanciu generátora kritérií pokrytia, podľa vstupných parametrov Generátora.

Modul generator

Hlavný modul, ktorý vo funkcii *generate_runs()* implementuje algoritmus 4.2 pre vygenerovanie testovacích behov. Ku tomu využíva funkcie a metódy popísane vyššie. Do tejto funkcie vstupujú už vygenerované testovacie požiadavky od užívateľa a algoritmus generuje testovacie behy nad modelom s agregovanými hranami, ktorý využíva metódy popísané v module *paths* pre vytváranie testovacích behov.

V module sa nachádza funkcia *extend_runs()*, ktorá na vstupe požaduje jeden argument s testovacími behmi. Pre každý beh sa vytvorí karteziánsky súčin z agregovaných hrán, z ktorých sa testovací beh skladá a tým vznikne množina testovacích behov skladajúca sa z jednotlivých prechodov modelu GUI. Formálne pre agregujúce hrany $AE_1, \ldots, AE_n$ je jeden beh dĺžky n definovaný ako:

$$TR = \{AE_1, \ldots, AE_n\}$$

a pre tento beh je vytvorená množina testovacích behov:

$$TRs = AE_1 \times \cdots \times AE_n = \{(e_1, \ldots, e_n) \mid e_i \in AE_i, 1 \leq i \leq n\}.$$

Užívateľ má možnosť na vstupe špecifikovať formátovanie pre jednotlivé behy. Môže zadať formát pre jednotlivé používateľské udalosti, môže zadať prefix a sufix každého testovacieho behu a zároveň je možné špecifikovať prefix celého súboru. Takýmto spôsobom má užívateľ možnosť predgenerovať testovaciu sadu pre konkrétny *xUnit framework*, ktorú je možné hneď spúšťať, popr. užívateľ ju môže upraviť – pridať asertáciu apod. O formátovanie testovacej sady sa stará funkcia *format_output()*

4.5.3 Behaviorálny popis Generátora

V tejto časti je popísaná komunikácia medzi jednotlivými časťami nástroja Generátor, ktoré boli popísané v predchádzajúcej kapitole. V prvom kroku sú spracované vstupné argumenty nástroja od používateľa v module *parsers*. Tento modul najprv overí správnosť argumentov a pripraví konfiguráciu na kritéria pokrytia pre generovanie testovacích požiadaviek a konfiguráciu pre formátovanie výstupu. Následne je deserializovaný vstupný model a pri tejto

deserializácii sa agregujú jednotlivé prechody medzi stavmi. Po spracovaní modelu, pomocou lexikálneho a syntaktického analyzátora sa spracujú užívateľom definované testovacie požiadavky.

Na základe konfigurácie sa vytvorí potrebná inštancia generátora pre testovacie požiadavky pomocou triedy *CoverageFactory()* a pomocou tohoto generátora sa vytvorí zoznam testovacích požiadaviek. Tieto testovacie požiadavky sú tvorené referenciami na hrany typu *AggregatedEdge*. Tento zoznam je predaný funkcii *generate_runs()*, ktorý vygeneruje abstraktné testovacie behy. Tieto abstraktné behy sú predané funkcii *extend_runs()*, ktorá pomocou karteziánskeho súčinu, popísaného v predchádzajúcej kapitole, sú vytvorené špecifické testovacie behy.

V prípade, že užívateľ definuje vlastné testovacie požiadavky, tak vygenerovaný zoznam so špecifickými testovacími behmi a užívateľské požiadavky sú znova predané funkcii *generate_runs()*, ktorá zopakuje priebeh algoritmu s poskytnutými požiadavkami a aktuálne nájdenými testovacími behmi. Po dobehnutí algoritmu sú výsledné behy predané funkcii *format_output()*, ktorá vytvorí testovaciu sadu na základe užívateľských preferencií, ktoré sú dostupné z konfigurácie.

Kapitola 5

Implementačné detaily

Táto kapitola obsahuje implementačné detaily niektorých častí systému. Je tu ukázané implementované rozhranie obidvoch nástrojov, ktoré môže využívať užívateľ, popr. iné nástroje z platformy *Testos*. Je tu ukázaná implementácia rozhrania *BaseDriver* pomocou knižnice *Selenium*, ktorá sa využíva na automatizáciu prehliadača a je tu ukázaný algoritmus na generovanie testovacích požiadaviek.

Celý navrhnutý systém bol implementovaný v jazyku Python vo verzii *3.7.3* a bol striedavo vyvýjaný na operačnom systéme Windows 10 a Ubuntu 18.04. Na úpravu zdrojových kódov bol používaný nástroj Visual Studio Code, rozšírený o balíček pre jazyk Python, ktorý umožňuje formátovať kód, napovedanie príkazov alebo ladenie programu.

V nástrojoch Monitor aj Generátor bolo okrem štandardných knižníc jazyka, použitá knižnica *ply*, ktorá je implementáciou pre nástroje *lex* a *yacc*. Monitor aj Generátor fungujú ako serverové služby, ktoré majú implementované rozhranie *REST API* pomocou knižnice *flask*. Monitor využíva knižnicu *Selenium* pre automatizáciu webového prehliadača, pomocou ktorej je implementované rozhranie *BaseDriver*.

5.1 Rozhrania implementovaných nástrojov

Jednou z požiadaviek bola integrácia nástroja do platformy *Testos*. Preto jednotlivé nástroje sú implementované ako serverové služby, ktoré majú rozhranie typu *REST API* implementované vďaka knižnici *flask*. Tieto služby prijímajú vstupne požiadavky vo formáte *JSON*, ktorý je definovaný pre každý endpoint rozhrania. Odpoveď prichádzajúca zo služby je tiež vo formáte *JSON* a skladá sa z dvoch položiek `"status"` a `"data"`. Tento formát platí aj pre implementovaný Monitor a aj Generátor Položka `"status"` obsahuje príznak, či sa podarilo vyhodnotiť požiadavku od klienta. Ak všetko prebehlo bez problémov, tak v položke `"data"` je obsiahnutá naformátovaná odpoveď pre klienta. Ak sa z nejakého dôvodu nepodarilo vyhodnotiť klientsku požiadavku, tak v odpovedi v položke `"data"` sa nachádza chybová hláška.

Hlavný cyklus Monitoru pre získanie modelu z GUI môže trvať niekoľko minút, v závislosti od rozsahu testovanej aplikácie. Z tohoto dôvodu celá funkcionalita nástroja je spúšťaná v osobitnom procese, aby klientske požiadavky pre rozhranie *REST API* boli čo najmenej blokujúce. Služba Monitor sa spúšťa pomocou aplikácie `monitor-api`, ktorá spustí hlavný cyklus z knižnice *flask*. Spustenie tohoto cyklu spolu s definovanými jednotlivými koncovými bodmi aplikácie sa nachádzajú v súbore `api.py`. Aplikácia Monitor ma implementované nasledujúce koncové body:

- `/start`, pomocou ktorého je spustená jedna inštancia Monitoru na základe poskytnutej konfigurácie a je vygenerovaný identifikátor, pomocou, ktorého je možné pristupovať na danú inštanciu pomocou ďalších koncových bodov,
- `/pause`, pomocou ktorého je možné pozastaviť výpočet Monitoru,
- `/end`, ukončuje špecifikovanú inštanciu Monitoru,
- `/is_running`, vracia v odpovedi príznak, či je Monitor aktuálne spustený,
- `/is_end`, vracia v odpovedi príznak, či Monitor dokončil získavanie,
- `/move_event`, je na zmenu poradia používateľských udalostí v rámci špecifikovaného stavu a slúži na zmenu behu monitora, už v priebehu získavania modelu z GUI,
- `/append_event`, je na pridanie novej používateľskej udalosti pre špecifikovaný stav a tiež slúži na zmenu behu monitora,
- `/get_model`, vracia doposiaľ objavený model GUI,
- `/get_events`, v odpovedi je vrátený zoznam vykonaných používateľských udalostí pre jednotlivé stavy modelu GUI.

Generátor nemá funkcionalitu oddelenú do separátneho procesu ako Monitor, pretože generovanie testovacích behov už nie je tak časovo náročný proces. Serverová služba generátoru sa spúšťa pomocou programu `generator-api` a rozhranie tejto aplikácie je implementované v súbore `api.py`. Aplikácia má iba jeden implementovaný koncový bod na adrese `/generate`, na ktorej prijíma požiadavok v tvare *JSON*, ktorý obsahuje vytvorený model a užívateľom špecifikované kritéria pokrytia a formátovanie výstupu.

Pre serverové rozhrania obidvoch implementovaných aplikácií platí, že je možné nastavovať nasledujúce premenné prostredia nasledovne:

- `MONITOR_API_HOST`, resp. `GENERATOR_API_HOST`, pre nastavenie názvu hostiteľa, pre ktorý počáva rozhranie REST API, ktorý je počiatočne nastavený na IP adresu ”0.0.0.0”,
- `MONITOR_API_PORT`, resp. `GENERATOR_API_PORT`, pre nastavenie portu, na ktorom počúva rozhranie REST API, ktorý je počiatočne nastavený na ”7000”
- `MONITOR_API_DEBUG`, resp. `GENERATOR_API_DEBUG`, pre nastavenie príznaku `"true"` alebo `"false"`, podľa toho či sa majú vypisovať ladiace výstupy nástroja, ktorých výpis je štandardne zakázaný.

Služby Monitora a Generátora majú vytvorené vlastné definície pre systém *Docker*. Z týchto definícií je možné vytvoriť *Docker* obrazy, ktoré užívateľ môže podľa potreby inštanciovať do kontajnerov, ktoré spravuje táto služba.

Okrem rozhrania *REST API* majú nástroje aj klasické rozhranie z príkazovej riadky. Parametre pre spustenie z príkazovej riadky sú pre jednotlivé nástroje popísané v prílohe B a C. Spustenie konzolovej aplikácie `monitor-cli` dokáže spustiť iba jeden proces, v ktorom je spustený Monitor. Pre spustenie ďalšej nezávislej inštancie Monitora je teda nutné spustiť ďalšiu konzolovú aplikáciu `monitor-cli`.

5.2 Použitie nástrojov z projektu Selenium

Táto časť obsahuje popis použitia rámca *Selenium* ako ovládača GUI pre webové aplikácie v tomto projekte a jeho integráciu s nástrojom Monitor. V návrhu 4.4.2 je popísané rozhranie *BaseDriver*, ktoré by mal implementovať každý ovládať GUI, aby ho bolo možné používať v module monitor. Jazyku Python nepodporuje možnosť definovať rozhranie, ale je možné použiť štandardný modul *abc*, pre definovanie abstraktných tried. Navrhnuté rozhranie *BaseDriver* tento modul nevyužíva, ale je implementované ako trieda, ktorá obsahuje definície pre navrhnuté metódy, ktoré keď sa zavolajú, tak je vyvolaná výnimka `NotImplementedError`.

Ovládač Selenium pre Monitor je implementovaný v triede *SeleniumDriver*, ktorá sa nachádza v module *drivers*. Na základe užívateľskej konfigurácie je pri vytváraní inštancie tohoto objektu, vytvorená inštancia definovaného prehliadača, ktorý sa spustí priamo na hostiteľskom počítači, alebo ak je zadaný parameter `selenium_hub_url`, tak sa spustí inštancia triedy *Remote* z modulu Selenium. V parametri `selenium_hub_url` musí byť správne špecifikovaná URL na vopred spustený Selenium Server. Ak je Monitor spúšťaný ako služba v kontajneri, tak je možné použiť možnosť iba s napojením na Selenium Server.

Objekt *SeleniumDriver* obsahuje premennú *_inner_driver* kde si uchováva referenciu na vytvorený ovládač z modulu *webdrivers* z knižnice Selenium. Nad touto referenciou, ovládač *SeleniumDriver* zapúzdruje volania jednotlivých metód na získanie prvkov pomocou selektorov, na získanie udalostí pre daný prvok a na vygenerovanie unikátneho selektoru v rámci stavu. Knižnica Selenium dokáže spúšťať kód v jazyku JavaScript na strane prehliadača. Pomocou tejto funkcionality je zabezpečené generovanie unikátneho selektoru pre ovládací prvok. Trieda *SeleniumDriver* má definovaný reťazec s predpripravenou funkciou, ktorej hlavný cyklus je vidieť v algoritme 5.1 a ktorá sa stará o vygenerovanie selektoru, postupným pripájaním selektorov rodičovských prvkov, kým sa nevyhodnotí vytvorený selektor ako jediný v danom stave GUI pomocou funkcie *isNotUnique()*.

```
1  function generateSelector(el){
2      selector = getNodeSelector(el);
3      parent = getElementsByXpath("..", el).snapshotItem(0);
4
5      while(isNotUnique("//" + selector)){
6          parent_selector = getNodeSelector(parent);
7
8          if (parent.tagName == 'body'){
9              return null;
10         }
11         selector = parent_selector + "/" + selector
12         parent = getElementsByXpath("..", parent).snapshotItem(0);
13
14     }
15     selector = "//" + selector
16     return selector;
17 }
```

Alg. 5.1: Implementovaná metóda v jazyku JavaScript na vygenerovanie unikátneho xPath selektoru pre ovládací prvok v triede *SeleniumDriver* v module *drivers*.

Funkcia *getNodeSelector()* vracia selektor pre element predaný argumentom podľa nasledujúcich pravidiel:

- základ selektoru vytvor podľa mena elementu,
- ak má element atribút `id`, tak ho pridaj do selektoru, inak
- ak má element atribút `class`, tak ho pridaj do selektoru, inak
- ak existuje element na rovnakej úrovni, tak pridaj do selektoru index aktuálneho elementu.

5.3 Generovanie testovacích požiadaviek

Algoritmus pre generovanie testovacích behov bol navrhnutý v kapitole 4.5 a na základe tohoto algoritmu vznikla implementácia a nachádza sa v súbore *generator.py*. Pre vygenerovanie behov je potrebné najprv pripraviť testovacie požiadavky. Generovanie testovacích požiadaviek sa pre každé kritérium pokrytia v priebehu implementácie niekoľko krát zmenilo. Prvotná implementácia jednotlivých tried na generovanie kritérií pokrytia obsahovala pre každú triedu zvlášť implementované prehľadávanie modelu a vytváranie požiadaviek. Nevýhodou tejto implementácie bola nutnosť upraviť každú implementovanú metódu *generate()* z rozhrania *BaseCoverage*, v prípade zmien v iných častiach nástroja, najmä v module *model.py*. Preto vznikla trieda *CustomCoverage*, v ktorej je implementovaný algoritmus 5.2. Tento algoritmus je schopný generovať testovacie požiadavky na základe dvoch parametrov, na základe poskytnutej dĺžky a na základe príznaku, či sa má jednať o jednoduché cesty a generuje požiadavky iba na pokrytie grafu.

```
1  def generate(self, model):
2     test_requirements = []
3     to_expand = model.model.get_edges()
4
5     while to_expand:
6        expanded_paths = []
7        path = to_expand.pop(0)
8
9        if self.length and len(path) == self.length:
10             test_requirements.append(TestRequirement(path))
11             continue
12
13       head = path[0]
14       tail = path[-1]
15
16       following = tail.get_following()
17       new_paths = [path + [i] for i in following if not self.only_simple_path
18             or i not in path]
19
20       if self.only_simple_path:
21             if head in following:
22                test_requirements.append(TestRequirement(path + [head]))
23
```

```
24        if new_paths:
25             to_expand += new_paths
26        elif self.only_simple_path:
27             test_requirements.append(TestRequirement(path))
28
29     return test_requirements
```

Alg. 5.2: Implementovaná metóda na vygenerovanie testovacích požiadaviek na základe užívateľských preferencií. Táto metóda sa nachádza v triede CustomCoverage v module coverages.

Kapitola 6

Overenie správnosti funkcionality systému a demonštrácia systému

Obsahom tejto kapitoly je popis jednotkových a integračných testov, ktorými je zabezpečené otestovanie dôležitých častí systému. Na implementáciu testov bola vybraná knižnica *pytest* napriek tomu, že jazyk Python má štandardnú knižnicu určenú na testovanie, zvanú *unittest*. Knižnica *pytest* bola vybraná najmä kvôli tomu, že má vlastné funkcie *assert*, ktoré v prípade zlyhania testu poskytujú mnoho informácií o stave SUT. Ďalším dôvodom bola pomerne jednoduchá príprava testovacieho prostredia pomocou *fixtures*.

Vytvorené testy sa nachádzajú v zložke `\tests`, ktorá obsahuje osobitnú zložku pre každý nástroj. V tejto zložke sú znova osobitne členené sady pre jednotkové a integračné testovanie. Každá takáto zložka má rovnaké členenie súborov a zložiek, ktoré je nasledovné:

- v súboroch `test_*.py` sa nachádzajú pripravené testovacie sady pre jednotlivé moduly nástroja, ktorých sa testy týkajú,
- v zložke `data` sú pripravené testovacie vstupné dáta, napr. konfiguračné súbory alebo vytvorené modely systému,
- súbor `fixtures.py` obsahuje funkcie, ktoré sú v testoch použité pre prípravu testovacieho prostredia pre testovanú jednotku do určitého stavu a
- konfiguračný súbor `conftest.py` slúži na registráciu vytvorených funkcií *fixture*, tak aby sa dali v testoch použiť z externého súboru.

Demonštrácia implementovaného systému je popísaná v kapitole 6.3, kde je popísané ako bol systém využitý na otestovanie webovej aplikácie *Calculatoria*[1], ktorá slúži ako vedecká kalkulačka. Pre túto aplikáciu bol vytvorený jej model po niekoľkých spusteniach Monitora, a následne bolo vytvorených niekoľko sád testovacích behov, ktoré spĺňajú rôzne kritéria pokrytia. Výsledné testovacie behy je možno vidieť v prílohe D.

6.1 Jednotkové testovanie

V tejto kapitole sa nachádza popis niektorých implementovaných testovacích sád pre nástroje Monitor a Generátor. Sú tu bližšie predstavené testovacie prípady – popis vstupných

[1] http://calculatoria.com/

testovacích dát, očakávaných výstupov a jednotlivých krokov. Zároveň sú tu popísané vytvorené testovacie *fixtures*, ktoré sa využívajú pre obidva nástroje.

Pre nástroj Monitor boli vytvorené celkovo tri testovacie sady s jednotkovými testami. Jedna z nich pokrýva implementovaný analyzátor *EventParser* pre gramatiku 4.5. Pre pokrytie spracovávania používateľských udalostí implementované v triede *EventParser* boli vytvorené testovacie dáta, ktoré zahŕňajú:

- prázdny reťazec,
- validné používateľské udalosti nad existujúcimi prvkami modelu,
- viacnásobné použitie používateľskej udalosti za sebou,
- používateľské udalosti so syntaktickými chybami,
- používateľské udalosti so sémantickými chybami a
- používateľské udalosti s použitím špeciálnych parametrov pre nástroj *Combine*.

Aby bolo možné vytvoriť inštanciu *EventParser*, boli vytvorené aj jednoduché dáta simulujúce množinu stavov z modelu GUI a vstupné definície pre špeciálne parametre *Combine*. Testovanie spočívalo v zavolaní analyzátoru *EventParser* a overení výsledku, či vytvorená postupnosť používateľských operácií pozostáva z očakávaných inštancií *PseudoEvent*, zo správnym identifikátorom ovládacieho prvku a správnou udalosťou. Pri testovaní chybných vstupných dát bola využitá funkcionalita knižnice *pytest* na overenie správnosti vyvolanej výnimky, ako je ukázané v testovacom prípade 6.1.

```
1 @pytest.mark.parametrize('model_mock', state_with_widgets)
2 class TestsEventParser:
3     @pytest.mark.parametrize('event, exception', invalid_event_test_data)
4     def test_parse_invalid_event(self, model_mock, event, exception):
5         parser = EventParser(model_mock['state_1'], [])
6         with pytest.raises(exception) as error:
7                 parser.parse_str(event)
```

Alg. 6.1: Príklad testovacieho prípadu pre parser *EventParser*, ktorý overuje, či pri spracovaní chybného vstupu je vyvolaná správna výnimka. Na prvom riadku sa pomocou anotácie triedy pripraví vstupný parameter, ktorý musí prijímať každý testovací prípad v rámci danej triedy. Na treťom riadku je podobný prípad, s rozdielom, že sa jedná o dvojicu parametrov. Na riadkoch 6. a 7. je sekvencia príkazov, pomocou ktorej sa overuje, či pri zavolaní funkcii je vyvolaný správny typ výnimky.

Pre nástroj Generátor bolo vytvorených šesť testovacích sád s jednotkovými testami. Jednou z nich je testovacia sada `test_tr_generators.py`, ktorá pokrýva požiadavku na systém REQ.12. Pre túto testovaciu sadu boli pripravené dva modely pre generovanie požiadaviek a pre každý typ požiadaviek boli ručne predpripravené očakávané výstupné dáta. Jeden model je acyklický a obsahuje štyri stavy, z toho dva počiatočné stavy, medzi ktorými existujú viacnásobné hrany a druhý model obsahuje vetvenie a jeden cyklus.

Jednotlivé modely sa nachádzajú v súboroch v zložke `\data` a pred spustením každého testu je potrebné daný model deserializovať. Aby sa to nemuselo vykonávať v rámci testu, tak je pre to nachystaný testovací *fixture*, ktorý je možné použiť v každom teste, ktorý má potrebnú anotáciu, ako je ukázané v ukážke 6.2. Očakávané výstupné dáta pre túto

testovaciu sadu sú v podobe množiny zoznamov, ktoré obsahujú identifikátory hrán. Na prevod identifikátorov na konkrétne inštancie hrán je tiež použitý *fixuture*.

```
1  @pytest.fixture
2  def model_filename_fixture():
3      return 'data'+os.sep+'model_simple.json'
4
5  @pytest.fixture
6  def model_fixture(model_filename_fixture):
7          _filename = os.path.join(test_module_path, model_filename_fixture)
8          return Model.from_file(_filename)
9
10 @pytest.mark.usefixtures("model_fixture")
11 @pytest.mark.parametrize('model_filename_fixture', ['data'+os.sep+'cycle.json'])
12 def test_parse_model_states(model_fixture):
13     # unittest
```

Alg. 6.2: Príklad vytvorených fixtures použitých v jednotkových testoch pre Generátor. Na riadkoch 2 a 6 sú definované fixtures, pre deserializáciu modelu zo súboru. Fixture `model_fixture` prijíma jeden parameter, čo je práve druhý definovaný fixture `model_filename_fixture` na riadku 2. Nad testom je následne definovaný iba jeden fixture, pomocou anotácie `usefixture`, ktorému je možné zmeniť štandardný parameter pomocou anotácie `parametrize`. Test môže využívať parameter *model_fixture*, v ktorom je už deserializovaný model.

6.2 Integračné testovanie

Nástroj Monitor aj Generátor obsahujú niekoľko integrácií, ktoré museli podliehať integračnému testovaniu. Táto časť obsahuje zoznam testovaných integrácií a detailnejší popis integračného testovania pre REST rozhranie nástroja Generátor – popis vstupných a očakávaných dát, jednotlivé kroky testov a popis použitých *fixtures*.

Integračné testovanie slúži na overenie správnosti komunikácie medzi jednotlivými časťami aplikácie alebo medzi dvoma aplikáciami. Pre nástroj Monitor boli pripravené testy na tieto integrácie:

- na rozhranie REST API, pre užívateľa alebo iné nástroje,
- na rozhranie medzi konkrétne vytvoreným ovládačom GUI a užívateľským rozhraním,
- na integráciu iterátora *EventIterator* s používateľskými udalosťami s nástrojom *Combine* z platformy *Testos*.

Pre nástroj Generátor boli vytvorené nasledujúce testovacie sady:

- na rozhranie REST API, pre užívateľa alebo iné nástroje,
- na rozhranie medzi prijatou požiadavkou s konfiguráciou a vytvoreným konkrétnym generátorom pre testovacie požiadavky.

Integračné testy pre rozhranie REST boli vytvorené podobným spôsobom pre obdiva vytvorené nástroje. Ako bolo spomenuté, cieľom je otestovanie komunikácie medzi jednotlivými časťami, v tomto prípade či nástroj je schopný vôbec vyhodnotiť jednotlivé požiadavky

poslané klientom a správne ich vyhodnotiť, pričom sa neprikladá veľký dôraz pre výstupné dáta testovaného nástroja.

Rozhranie nástroja Generátor poskytuje jeden koncový bod `/generate`, na ktorom očakáva presne špecifikovaný JSON, v ktorom je zahrnutý model a informácie pre vygenerovanie testovacích behov – kritéria pokrytia a dodatočné testovacie požiadavky. Funkcionalita generovania požiadaviek a behov je overená pomocou jednotkových testov. Pri integračnom testovaní prebehne funkcionalita jednotlivých častí systému, ale na vzorke vytvorených testovacích dát je testované iba spojenie medzi klientom a Generátorom, tzn.:

- či server je schopný odpovedať očakávaným návratovým statusom, pre správne dáta, ale aj dáta so syntaktickými a sémantickými chybami – chýbajúci model, syntaktické a sémantické chyby v dodatočných požiadavkách,
- či server je schopný spracovať viac požiadaviek zároveň, obsahujúce správne aj chybné dáta, popr. ich kombináciu.

```
1  def generator_server(xprocess):
2          class GeneratorStarter(ProcessStarter):
3                  env = os.environ.copy()
4                  env['GENERATOR_API_HOST'] = '127.0.0.1'
5                  env['GENERATOR_API_PORT'] = '7700'
6                  env['GENERATOR_API_DEBUG'] = 'tRuE'
7
8                  pattern = ".*Running on.*"
9                  args = [sys.executable, "-m", 'generator.api']
10
11         pid, log = xprocess.ensure("generator_api", GeneratorStarter)
12
13         url = "http://127.0.0.1:7700/generate"
14         request = urllib.request.Request(URL, method="GET")
15
16         yield request
17
18         process_info = xprocess.getinfo("generator_api")
19         process_info.terminate()
```

Alg. 6.3: Vytvorený testovací fixture, ktorý slúži na spustenie rozhrania REST nástroja Generátor pomocou knižnice `python-xprocess`. Na riadku 14 je vykonané spustenie pomocou predpisu definovaného v triede na riadku 2. Predpis obsahuje aj premennú `pattern`, pomocou ktorej sa špecifikuje, pri akom reťazci na výstupe nástroja sa považuje nástroj za spustený. Na riadku 16 sa predáva vytvorený predpis požiadavku, ktorý môže testovací prípad použiť na komunikáciu s vytvorenou inštanciou serveru. Na riadkoch 18, 19 je operácia zastavenia vytvoreného serveru po vykonaní testovacieho prípadu.

Pre vykonanie testovacích krokov je potrebné mať spustenú inštanciu Generátora, tzn. že nástroj musí byť paralelne spustený spolu s vykonávaným testovacím prípadom a zároveň pred začatím testovacieho behu musí byť server v stave, v ktorom môže prijímať požiadavky (nemôže sa nachádzať vo fáze inicializácie). Pre splnenie týchto požiadaviek je použitá knižnica `pytest-xprocess`, ktorá rozširuje `pytest` o možnosť vytvárania testovacích *fixtures*, ktoré sú v podobe paralelných procesov. Príklad takéhoto vytvoreného *fixture* je ukázaný v algoritme D.

6.3 Demonštrácia systému

V tejto časti je ukázaný príklad použitia implementovaného nástroja na generovanie behov nad webovou kalkulačkou, ktorej ukážku je vidieť na obrázku 6.1 a niektoré vygenerované testovacie behy sú priložené v prílohe D, a ďalšie dáta sú uložené na pamäťovom médiu – vygenerovaný model aplikácie, všetky vygenerované testovacie behy a klientský skript pre vygenerovanie testovacej sady, ktoré je možné spustiť pomocou testovacieho rámca *pytest*.

Vytváranie testovacích prípadov bolo zamerané výhradne na funkcionalitu danej kalkulačky, pretože aplikácia okrem iného obsahuje komponentu s výsledkovým blokom, ktorý je možné stiahnuť ako súbor, alebo vytlačiť, a zároveň sa na stránke nachádzajú rôzne generované reklamy.

Obrázek 6.1: Ukážka aplikácie, na ktorej je demonštrovaná funkcionalita implementovaného systému.

Pred prvým spustením Monitora na vygenerovanie modelu, bolo potreba manuálne zanalyzovať jednotlivé ovládacie prvky na testovanej stránke, aby mohli byť vyhodnotené selektory, ktoré sa dajú na vstup pre nástroj Monitor. Ovládacie prvky kalkulačky sa dajú generalizovať tromi selektormi:

- `//*[@id='kuldiv']//a` – tlačidlá kalkulačky, odkazy na štandardnú kalkulačku a tlačidlá na zmenu veľkosti kalkulačky,
- `//*[@id='kuldiv']//select` – komponenta na výber zaokrúhlenia výsledku a
- `//*[@id='kuldiv']//input` – formulárový vstup slúžiaci ako display kalkulačky.

Konfigurácia pre monitor obsahovala okrem selektorov, jeden špecifikovaný stav a špecifikovaný webový prehliadač, ktorý sa má použiť pre ovládač GUI. Po uložení modelu GUI do súboru, boli vygenerované testovacie sady pomocou pripraveného klienta pre nástroj Generátor. Vytvorený klient očakáva spustený nástroj Generátor na adrese a porte, ktoré je možné špecifikovať pomocou premenných prostredia. Klient na vstupe očakáva parametrom špecifikovaný model GUI, konfiguračný súbor pre generátor so špecifikovanými kritériami pokrytia a názov výstupného súboru pre výslednú testovaciu sadu. Po spracovaní vstupných dát, klient pošle požiadavku na server Generátora, a na základe vygenerovanej odpovede klient spracuje jednotlivé testovacie behy do výstupného súboru a v takom formáte, aby sa dali znova použiť pomocou knižnice *pytest*. Do tohoto súboru je zapracovaný testovací *fixture*, pre spúšťanie ovládača GUI s prehliadačom *Chrome*, ktorý sa používa pre jednotlivé testovacie prípady.

Pre predmetnú aplikáciu boli vytvorené štyri testovacie sady pokrývajúce nasledujúcu funkcionalitu kalkulačky:

- aritmetické operácie (+, -, *, /) kalkulačky sú pokryté testovacou sadou `TestSuiteAritmetic`,
- overenie pamäťových funkcií kalkulačky, zahŕňa testovacia sada `TestSuiteMemory`,
- funkcionalitu goniometrických funkcií pokrýva testovacia sada `TestSuiteTrigonometric` a
- ostatné pokročilé funkcie (faktoriál, logaritmus, mocniny) kalkulačky sú pokrytie testovacou sadou `TestSuiteFunctions`.

Všetky testovacie sady vychádzali z rovnakého kritéria pokrytia – pokrytia všetkých hrán. Pokrytie jednotlivej funkcionality vychádza z pripravených dodatočných testovacích požiadaviek, ktoré boli špecifikované ručne pomocou vytvoreného jazyka.

Kapitola 7

Záver

V rámci tejto práce boli prestavené metódy pre testovanie grafických používateľských rozhraní. Bližšie bola popísaná metóda testovania zo znalosti štruktúry GUI a systém *Selenium*, pomocou ktorého je možné testovať webové aplikácie. Ďalej bol popísaný proces testovania na základe modelu, pri ktorom boli zhodnotené jeho výhody a nevýhody. Bližšie boli popísané existujúce nástroje, ktoré využívajú princíp testovania na základe modelu a je možné ich využiť na testovanie GUI.

Na základe skúmania jednotlivých metód a naštudovaných nástrojov boli vykonaná analýza a spísané požiadavky na nástroj pre generovanie testovacích behov pre grafické používateľské rozhrania. Z vytvorených požiadaviek bol navrhnutý konceptuálny návrh, ktorý zachytáva hlavné časti systému, ich vstupy a výstupy a ukazuje celý priebeh testovania pomocou tohoto nástroja spolu s používateľom. Pre potreby modelovania GUI bol vytvorený formálny popis modelu, ktorý zachytáva dôležité aspekty testovaného systému. Spolu s týmto popisom bol ukázaný algoritmus pre získanie modelu zo systému s GUI pomocou postupného získavania informácií o ovládacích prvkoch a spolu s týmto algoritmom bol navrhnutý nástroj, pomocou, ktorého je možné automatizovanie vytvárať modely z GUI.

Druhou časťou návrhu bolo formálne popísanie testovacieho behu, ktorý vychádza z prechodov vytvoreného modelu. Pre testovacie behy boli definované kritéria pokrytia a jazyk, pomocou neho používateľ dokáže vytvárať vlastné testovacie požiadavky. Pre generovanie testovacích behov bol navrhnutý nástroj, ktorý vychádza z formálne popísaného algoritmu na generovanie behov.

Na základe návrhu boli jednotlivé nástroje implementované spolu s rozhraním REST API, ktoré slúži najmä pre integráciu s platformou *Testos*. Pre obidva nástroje boli zvlášť vytvorené jednotkové a integračné testy pre zaistenie overenia najdôležitejšej funkcionality. Bola vybraná testovacia aplikácia na demonštrovanie funkcionality vytvorenia testovacích behov. Pre tieto účely bol pripravený klient, ktorý dokáže z vygenerovaných testovacích behov vytvoriť spustiteľné testovacie sady.

Vývoj systému by mohol ďalej pokračovať v rozširovaní obidvoch nástrojov. Monitor GUI by v budúcnosti mohol implementovať ďalšie ovládače GUI, pre ktoré bolo vytvorené rozhranie. Zároveň by sa mohli vylepšiť schopnosti systému pri vyhodnocovaní rovnosti dvoch stavov GUI. Generátor by sa dal rozšíriť o viac možnosť generovaní testovacie požiadavky na základe ďalších kritérií pokrytia, napr. pokrytia pre dátové toky, čo by ale znamenalo zásah do používaného modelu.

Literatura

[1] Ammann, P.; Offutt, J.: *Introduction to Software Testing*. New York, NY, USA: Cambridge University Press, první vydání, 2008, ISBN 0521880386, 9780521880381.

[2] Contributors, S.: Selenium 2.0. online, 2011-07-8 [cit. 2019-05-15].
URL https://seleniumhq.wordpress.com/2011/07/08/selenium-2-0/

[3] Contributors, S.: Selenium 3.0. online, 2016-10-13 [cit. 2019-05-15].
URL https://seleniumhq.wordpress.com/2016/10/13/selenium-3-0-out-now/

[4] Contributors, S.: Selenium Remote Control. online, 2019-05-14 [cit. 2019-05-15].
URL https://www.seleniumhq.org/docs/05_selenium_rc.jsp

[5] Contributors, S.: Selenium History. online, [cit. 2019-05-15].
URL https://www.seleniumhq.org/about/history.jsp

[6] Dallmeier, V.; Burger, M.; Orth, T.; aj.: WebMate: Generating Test Cases for Web 2.0. In *Software Quality. Increasing Value in Software and Systems Development*, editace D. Winkler; S. Biffl; J. Bergsmann, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN 978-3-642-35702-2, s. 55–69.

[7] Dallmeier, V.; Pohl, B.; Burger, M.; aj.: WebMate: Web Application Test Generation in the Real World. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, March 2014, s. 413–418, doi:10.1109/ICSTW.2014.65.

[8] Galitz, W. O.: *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. New York, NY, USA: John Wiley & Sons, Inc., 2007, ISBN 0470053429.

[9] Hambling, ., Brian; Morgan, P.; Society, B. C.: *Software testing : an ISTQB-ISEB foundation guide*. London : British Computer Society, druhé vydání, 2010, ISBN 9781906124762 (pbk.), previous ed.: 2007.

[10] Jorgensen, P.: *The Craft of Model-Based Testing*. CRC Press/Taylor & Francis Group, 2017, ISBN 9781498712286.
URL https://books.google.sk/books?id=85T5vQAACAAJ

[11] Kramer, A.; Legeard, B.: *Model-Based Testing Essentials: Guide to the ISTQB® Certified Model-Based Tester Foundation Level*. 04 2016, 1-276 s., doi:10.1002/9781119130161.

[12] Memon, A.; Banerjee, I.; Nagarajan, A.: GUI ripping: reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, Nov 2003, ISSN 1095-1350, s. 260–269, doi:10.1109/WCRE.2003.1287256.

[13] Memon, A. M.; Soffa, M. L.; Pollack, M. E.: Coverage Criteria for GUI Testing. *SIGSOFT Softw. Eng. Notes*, ročník 26, č. 5, Září 2001: s. 256–267, ISSN 0163-5948, doi:10.1145/503271.503244.
URL http://doi.acm.org/10.1145/503271.503244

[14] Mesbah, A.; van Deursen, A.: Invariant-based Automatic Testing of AJAX User Interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, Washington, DC, USA: IEEE Computer Society, 2009, ISBN 978-1-4244-3453-4, s. 210–220, doi:10.1109/ICSE.2009.5070522.
URL http://dx.doi.org/10.1109/ICSE.2009.5070522

[15] Nguyen, B. N.; Robbins, B.; Banerjee, I.; aj.: GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, ročník 21, č. 1, Mar 2014: s. 65–105, ISSN 1573-7535, doi:10.1007/s10515-013-0128-9.
URL https://doi.org/10.1007/s10515-013-0128-9

[16] Pinto, M.; Gonçalves, M.; Masci, P.; aj.: TOM: A Model-Based GUI Testing Framework. In *Formal Aspects of Component Software*, editace J. Proença; M. Lumpe, Cham: Springer International Publishing, 2017, ISBN 978-3-319-68034-7, s. 155–161.

[17] Preece, J.; Rogers, Y.; Sharp, H.: *Interaction Design*. John Wiley & Sons, Inc., třetí vydání, 2011, ISBN 978-0-470-66576-3.

[18] Pretschner, A.; Philipps, J.: *10 Methodological Issues in Model-Based Testing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ISBN 978-3-540-32037-1, s. 281–291, doi:10.1007/11498490_13.
URL https://doi.org/10.1007/11498490_13

[19] Sommerville, I.: *Software Engineering*. Pearson, 10 vydání, 2015, ISBN 0133943038, 9780133943030.

[20] Utting, M.; Legeard, B.: *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, ISBN 0123725011, 9780080466484.

[21] Utting, M.; Pretschner, A.; Legeard, B.: A Taxonomy of Model-based Testing Approaches. *Softw. Test. Verif. Reliab.*, ročník 22, č. 5, Srpen 2012: s. 297–312, ISSN 0960-0833, doi:10.1002/stvr.456.
URL http://dx.doi.org/10.1002/stvr.456

[22] Xu, D.: A Tool for Automated Test Code Generation from High-Level Petri Nets. In *Applications and Theory of Petri Nets*, editace L. M. Kristensen; L. Petrucci, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ISBN 978-3-642-21834-7, s. 308–317.

[23] Xu, D.; Xu, W.; Kent, M.; aj.: An Automated Test Generation Technique for Software Quality Assurance. *IEEE Transactions on Reliability*, ročník 64, č. 1, March 2015: s. 247–268, ISSN 0018-9529, doi:10.1109/TR.2014.2354172.

Příloha A

Obsah pamäťového média

Pamäťové médium obsahuje nasledujúce súbory:

- **doc** - zložka so zdrojovými kódmi tejto textovej dokumentácie,
- **demo** - zložka obsahuje konfiguračné súbory pre nástroje Monitor a Generátor použité pre vygenerovanie testovacích behov, vygenerované testovacie behy a nástroj pre opakované spustenie vytvorených testovacích behov,
- **src** - zdrojové kódy ku vytvoreným nástrojom,
- **tests** - zdrojové kódy testov k implementovaným nástrojom,
- **README.MD** - informácie k súborom a návod ku spusteniu nástrojov,
- **Makefile**.

Příloha B

Manuál ku spusteniu nástroja Monitor

Zoznam jednotlivých parametrov aplikácie konzolovej aplikácie Monitor:

- `-d`, `-driver` – na zvolenie ovládača GUI, je možné zvoliť iba "selenium"(povinný),
- `-c`, `-config` – špecifikuje sa konfiguračný cesta ku konfiguračnému súboru pre ovládač GUI (povinný),
- `-m`, `-model_out` – cesta, kde sa má uložiť model (štandardne model.out.json),
- `-e`, `-events_out` – cesta, kde sa majú uložiť zaznamenané udalosti (štandardne events.out.yaml),
- `-u`, `-update_model` – príznak, či sa má načítať model a udalosti zo vstupu a daný model model aktualizovať,
- `-M`, `-model_in` – cesta k súboru, odkiaľ sa má načítať model pre aktualizáciu,
- `-E`, `-events_in` – cesta k súboru, odkiaľ sa majú načítať zaznamenané udalosti,
- `-C`, `-combine_url` – URL pre rozhranie Combine API-

1. Spustenie konzolovej aplikácie Monitor

```
$ make monitor
$ monitor-cli -d selenium -c demo/selenium_config.yaml
```

2. Spustenie Selenium Serveru v Docker kontajenri a spustenie Monitoru:

```
$ docker run -d -p 4444:4444 --shm-size=2g selenium/standalone-chrome

$ make monitor
$ monitor-cli -d selenium -c demo/selenium_config_remote.yaml
```

3. Spustenie Monitora ako REST API služby. API Monitoru je možné pomocou premenných prostredia nastaviť nasledovne:

- `MONITOR_API_HOST` – názov hostiteľa (štandardne 0.0.0.0),

- `MONITOR_API_PORT` – číslo portu (štandardne 7000) a
- `MONITOR_API_DEBUG` – príznak pri vypisovanie ladiacich výpisov.

```
$ make monitor
$ monitor-api
```

4. Spustenie Monitora ako Docker kontajneru s REST API. V definičnom súbore je možné nastaviť API pomocou úpravy premenných prostredia. Príklad spustenia kontajneru s monitorom, ktorý počúva na porte 8081:

```
# vytvorenie obrazu na zaklade Dockerfile a spustenie kontajneru
$ docker build -t monitor -f src/docker/monitor/Dockerfile src/monitor
$ docker run -p 8081:7000 monitor
```

Příloha C

Manuál ku spusteniu nástroja Generátor

1. Spustenie konzolovej aplikácie nástroja Generátor, ktorá má dva povinné parametre

   - `-m`, `-model` – model GUI, z nástroja Monitor,
   - `-c`, `-cc` – konfiguračný súbor pre generovanie a formátovanie behov.

```
$ make generator
$ generator-cli --model demo/calculatoria.json --cc demo/generator_in.json

# alebo spustenie so skratenymi argumentami
$ generator-cli -m demo/calculatoria.json -c demo/generator_in.json

# alebo spustenie spolu s ladiacimi vypismi
$ generator-cli -m demo/calculatoria.json -c demo/generator_in.json --debug
```

2. Spustenie Generátora ako REST API služby. API Generátoru je možné pomocou premenných prostredia nastaviť nasledovne:

   - `GENERATOR_API_HOST` – názov hostiteľa (štandardne 0.0.0.0),
   - `GENERATOR_API_PORT` – číslo portu (štandardne 7000) a
   - `GENERATOR_API_DEBUG` – príznak pri vypisovanie ladiacich výpisov.

```
$ make generator
$ generator-api

# alebo spustenie spolu s ladiacimi vypismi
$ generator-api -d
```

3. Spustenie Generátora ako Docker kontajneru s REST API. V definičnom súbore je možné nastaviť API pomocou úpravy premenných prostredia. Príklad spustenia kontajneru s generátorom, ktorý počúva na porte 8080:

```
# vytvorenie obrazu na zaklade Dockerfile a spustenie kontajneru
$ docker build -t generator -f src/docker/generator/Dockerfile src/generator
$ docker run -p 8080:7000 generator
```

Ovládanie klientskej aplikácie pre Generátor API

V zložke `demo` sa nachádza pripravený klient pre REST API nástroja generátor, ktorý na vstupe príma model GUI a názov výstupného súboru, do ktorého sa zapíšu testovacie behy. V súbore s klientom je možné upravovať formát výstupných testovacích behov `client_generator.py` a ďalšie informácie nutné pre generovanie behov. Klient číta z premenných prostredia `TC_GENERATE_HOST` a `TC_GENERATE_PORT`, hostiteľský názov počítača a port, na ktorom by mal byť spustený Generátor.

```
$ cd demo
$ python3 client_generator.py -m calculatoria.json -o test_suite.py
$ pytest test_suite.py
```

Príkaz pre spustenie vygenerovanej sady `pytest test_suite.py` spúšťa lokálne prehliadač Chrome, takže je potrebné mať nainštalovaný nástroj *chromedriver*, knižnice do jazyka Python *pytest* a *selenium*.

Knižnice pre spustenie demonštračnej testovacej sady je možné nainštalovať pomocou:

```
$ cd demo
$ pip3 install -r demo_requirements.txt
```

Příloha D

Príklad vygenerovaných testovacích behov pre nástroj Calculatoria

Príkaz volania `find_element_byq_xpath()` s knižnice Selenium bol pre nahradený za `f()` kvôli demonštračnej ukážke.

```
1  import pytest
2  from selenium import webdriver
3
4
5  @pytest.fixture(scope="class")
6  def chrome_driver_init(request):
7      chrome_driver = webdriver.Chrome()
8      request.cls.driver = chrome_driver
9      yield
10     chrome_driver.close()
11
12
13 @pytest.mark.usefixtures("chrome_driver_init")
14 class TestSuiteGenerated():
15   def test_case_0(self):
16     self.driver.get("http://www.calculatoria.com/")
17     self._d.f("//div[25][@class='btnmar btn2'] \
18         /a[1][@class='abtn2 zabtn']").click()
19     self._d.f("//div[25][@class='btnmar btn2'] \
20         /a[1][@class='abtn2 zabtn']").click()
21     self._d.f("//div[27][@id='btn110'][@class='btnmar btn2'] \
22         /a[1][@class='abtn2 zabtn']").click()
23
24   def test_case_1(self):
25     self.driver.get("http://www.calculatoria.com/")
26     self._d.f("//div[24][@id='btn109'][@class='btnmar btn1'] \
27         /a[1][@class='abtn1 zabtn']").click()
28     self._d.f("//div[3][@class='btnmar btn2'] \
29         /a[1][@class='abtn2 zabtn']").click()
30     self._d.f("//div[3][@class='btnmar btn2'] \
```

```
31          /a[1][@class='abtn2 zabtn']").click()
32
33    def test_case_2(self):
34      self.driver.get("http://www.calculatoria.com/")
35      self._d.f("//div[23][@id='btn99'][@class='btnmar btn2'] \
36          /a[1][@class='abtn2 zabtn']").click()
37      self._d.f("//div[3][@class='btnmar btn2'] \
38          /a[1][@class='abtn2 zabtn']").click()
39      self._d.f("//div[3][@class='btnmar btn2'] \
40          /a[1][@class='abtn2 zabtn']").click()
41
42    def test_case_3(self):
43      self.driver.get("http://www.calculatoria.com/")
44      self._d.f("//div[22][@id='btn98'][@class='btnmar btn2'] \
45          /a[1][@class='abtn2 zabtn']").click()
46      self._d.f("//div[3][@class='btnmar btn2'] \
47          /a[1][@class='abtn2 zabtn']").click()
48      self._d.f("//div[3][@class='btnmar btn2'] \
49          /a[1][@class='abtn2 zabtn']").click()
50      self._d.f("//div[26][@id='btn96'][@class='btnmar btn2'] \
51          /a[1][@class='abtn2 zabtn']").click()
52      self._d.f("//div[20][@class='btnmar btn2'] \
53          /a[1][@class='abtn2 zabtn']").click()
54      self._d.f("//div[14][@id='btn111'][@class='btnmar btn1'] \
55          /a[1][@class='abtn1 zabtn']").click()
56      self._d.f("//div[3][@class='btnmar btn2'] \
57          /a[1][@class='abtn2 zabtn']").click()
58      self._d.f("//div[26][@id='btn96'][@class='btnmar btn2'] \
59          /a[1][@class='abtn2 zabtn']").click()
60      self._d.f("//div[20][@class='btnmar btn2'] \
61          /a[1][@class='abtn2 zabtn']").click()
62      self._d.f("//div[14][@id='btn111'][@class='btnmar btn1'] \
63          /a[1][@class='abtn1 zabtn']").click()
64      self._d.f("//div[20][@class='btnmar btn2']/ \
65              a[1]/[@class='abtn2 zabtn']").click()
66      self._d.f("//div[14][@id='btn111'][@class='btnmar btn1'] \
67          /a[1][@class='abtn1 zabtn']").click()
68      self._d.f("//a[1][@class='abtn3p zabtn']").click()
69      self._d.f("//div[14][@id='btn111'][@class='btnmar btn1'] \
70          /a[1][@class='abtn1 zabtn']").click()
71      self._d.f("//a[1][@class='abtn3p zabtn']").click()
72      self._d.f("//div[14][@id='btn111'][@class='btnmar btn1'] \
73          /a[1][@class='abtn1 zabtn']").click()
74
75    def test_case_4(self):
76      self.driver.get("http://www.calculatoria.com/")
77      self._d.f("//div[20][@class='btnmar btn2'] \
78          /a[1][@class='abtn2 zabtn']").click()
```

```
79      self._d.f("//div[20][@class='btnmar btn2'] \
80          /a[1][@class='abtn2 zabtn']").click()
81      self._d.f("//div[14][@id='btn111'][@class='btnmar btn1'] \
82          /a[1][@class='abtn1 zabtn']").click()
83
84    def test_case_5(self):
85      self.driver.get("http://www.calculatoria.com/")
86      self._d.f("//div[19][@id='btn106'][@class='btnmar btn1'] \
87          /a[1][@class='abtn1 zabtn']").click()
88      self._d.f("//div[11][@id='btn103'][@class='btnmar btn2'] \
89          /a[1][@class='abtn2 zabtn']").click()
90      self._d.f("//div[19][@id='btn106'][@class='btnmar btn1'] \
91          /a[1][@class='abtn1 zabtn']").click()
92      self._d.f("//div[11][@id='btn103'][@class='btnmar btn2'] \
93          /a[1][@class='abtn2 zabtn']").click()
94      self._d.f("//a[1][@class='abtn2 zabtn bo']").click()
95      self._d.f("//div[18][@id='btn102'][@class='btnmar btn2'] \
96          /a[1][@class='abtn2 zabtn']").click()
97
98    def test_case_6(self):
99      self.driver.get("http://www.calculatoria.com/")
100     self._d.f("//div[17][@id='btn101'][@class='btnmar btn2'] \
101         /a[1][@class='abtn2 zabtn']").click()
102     self._d.f("//div[17][@id='btn101'][@class='btnmar btn2'] \
103         /a[1][@class='abtn2 zabtn']").click()
104     self._d.f("//a[1][@class='abtn2 zabtn bo']").click()
105
106   def test_case_7(self):
107     self.driver.get("http://www.calculatoria.com/")
108     self._d.f("//div[16][@id='btn100'][@class='btnmar btn2'] \
109         /a[1][@class='abtn2 zabtn']").click()
110     self._d.f("//div[9][@id='btn83'][@class='btnmar btn2'] \
111         /a[1][@class='abtn2 zabtn']").click()
112     self._d.f("//div[9][@id='btn83'][@class='btnmar btn2'] \
113         /a[1][@class='abtn2 zabtn']").click()
114     self._d.f("//div[14][@id='btn111'][@class='btnmar btn1'] \
115         /a[1][@class='abtn1 zabtn']").click()
116
117   def test_case_8(self):
118     self.driver.get("http://www.calculatoria.com/")
119     self._d.f("//div[15][@class='btnmar btn2'] \
120         /a[1][@class='abtn2 zabtn']").click()
121     self._d.f("//div[15][@class='btnmar btn2'] \
122         /a[1][@class='abtn2 zabtn']").click()
123     self._d.f("//div[27][@id='btn110'][@class='btnmar btn2'] \
124         /a[1][@class='abtn2 zabtn']").click()
125     self._d.f("//div[27][@id='btn110'][@class='btnmar btn2'] \
126         /a[1][@class='abtn2 zabtn']").click()
```

```
127
128   def test_case_9(self):
129     self.driver.get("http://www.calculatoria.com/")
130     self._d.f("//div[13][@id='btn105'][@class='btnmar btn2'] \
131         /a[1][@class='abtn2 zabtn']").click()
132     self._d.f("//div[13][@id='btn105'][@class='btnmar btn2'] \
133         /a[1][@class='abtn2 zabtn']").click()
134     self._d.f("//a[1][@class='abtn2 zabtn bo']").click()
135     self._d.f("//div[18][@id='btn102'][@class='btnmar btn2'] \
136         /a[1][@class='abtn2 zabtn']").click()
137     self._d.f("//div[18][@id='btn102'][@class='btnmar btn2'] \
138         /a[1][@class='abtn2 zabtn']").click()
139     self._d.f("//div[21][@id='btn97'][@class='btnmar btn2'] \
140         /a[1][@class='abtn2 zabtn']").click()
141     self._d.f("//div[21][@id='btn97'][@class='btnmar btn2'] \
142         /a[1][@class='abtn2 zabtn']").click()
143     self._d.f("//div[10][@id='btn67'][@class='btnmar btn2'] \
144         /a[1][@class='abtn2 zabtn']").click()
145     self._d.f("//div[10][@id='btn67'][@class='btnmar btn2'] \
146         /a[1][@class='abtn2 zabtn']").click()
147
148   def test_case_10(self):
149     self.driver.get("http://www.calculatoria.com/")
150     self._d.f("//div[12][@id='btn104'][@class='btnmar btn2'] \
151         /a[1][@class='abtn2 zabtn']").click()
152     self._d.f("//div[12][@id='btn104'][@class='btnmar btn2'] \
153         /a[1][@class='abtn2 zabtn']").click()
154     self._d.f("//div[27][@id='btn110'][@class='btnmar btn2'] \
155         /a[1][@class='abtn2 zabtn']").click()
156
157   def test_case_11(self):
158     self.driver.get("http://www.calculatoria.com/")
159     self._d.f("//div[11][@id='btn103'][@class='btnmar btn2'] \
160         /a[1][@class='abtn2 zabtn']").click()
161     self._d.f("//div[3][@class='btnmar btn2'] \
162         /a[1][@class='abtn2 zabtn']").click()
163     self._d.f("//div[3][@class='btnmar btn2'] \
164         /a[1][@class='abtn2 zabtn']").click()
165
166   def test_case_12(self):
167     self.driver.get("http://www.calculatoria.com/")
168     self._d.f("//div[10][@id='btn67'][@class='btnmar btn2'] \
169         /a[1][@class='abtn2 zabtn']").click()
170     self._d.f("//div[10][@id='btn67'][@class='btnmar btn2'] \
171         /a[1][@class='abtn2 zabtn']").click()
172     self._d.f("//div[10][@id='btn67'][@class='btnmar btn2'] \
173         /a[1][@class='abtn2 zabtn']").click()
174     self._d.f("//div[10][@id='btn67'][@class='btnmar btn2'] \
```

```
175         /a[1][@class='abtn2 zabtn']").click()
176
177     def test_case_13(self):
178       self.driver.get("http://www.calculatoria.com/")
179       self._d.f("//div[8][@id='btn82'][@class='btnmar btn2'] \
180           /a[1][@class='abtn2 zabtn']").click()
181       self._d.f("//div[8][@id='btn82'][@class='btnmar btn2'] \
182           /a[1][@class='abtn2 zabtn']").click()
183       self._d.f("//a[1][@class='abtn6 zabtn']").click()
184       self._d.f("//a[1][@class='abtn6 zabtn']").click()
185
186     def test_case_14(self):
187       self.driver.get("http://www.calculatoria.com/")
188       self._d.f("//div[7][@id='btn73'][@class='btnmar btn2'] \
189           /a[1][@class='abtn2 zabtn']").click()
190       self._d.f("//div[7][@id='btn73'][@class='btnmar btn2'] \
191           /a[1][@class='abtn2 zabtn']").click()
192       self._d.f("//div[28][@id='btn13'][@class='btnmar btn2'] \
193           /a[1][@class='abtn2 zabtn']").click()
194       self._d.f("//div[28][@id='btn13'][@class='btnmar btn2'] \
195           /a[1][@class='abtn2 zabtn']").click()
196
197     def test_case_15(self):
198       self.driver.get("http://www.calculatoria.com/")
199       self._d.f("//div[6][@id='btn80'][@class='btnmar btn2'] \
200           /a[1][@class='abtn2 zabtn']").click()
201       self._d.f("//div[6][@id='btn80'][@class='btnmar btn2'] \
202           /a[1][@class='abtn2 zabtn']").click()
203       self._d.f("//div[27][@id='btn110'][@class='btnmar btn2'] \
204           /a[1][@class='abtn2 zabtn']").click()
205
206     def test_case_16(self):
207       self.driver.get("http://www.calculatoria.com/")
208       self._d.f("//a[1][@class='abtn2 zabtn bo']").click()
209       self._d.f("//a[1][@class='abtn2 zabtn bo']").click()
210       self._d.f("//div[28][@id='btn13'][@class='btnmar btn2'] \
211           /a[1][@class='abtn2 zabtn']").click()
212       self._d.f("//div[28][@id='btn13'][@class='btnmar btn2'] \
213           /a[1][@class='abtn2 zabtn']").click()
214       self._d.f("//div[28][@id='btn13'][@class='btnmar btn2'] \
215           /a[1][@class='abtn2 zabtn']").click()
216
217     def test_case_17(self):
218       self.driver.get("http://www.calculatoria.com/")
219       self._d.f("//div[2][@class='btnmar btn2'] \
220           /a[1][@class='abtn2 zabtn']").click()
221       self._d.f("//div[2][@class='btnmar btn2'] \
222           /a[1][@class='abtn2 zabtn']").click()
```

```
223     self._d.f("//a[1][@class='abtn6 zabtn']").click()
224     self._d.f("//a[1][@class='abtn6 zabtn']").click()
225     self._d.f("//div[28][@id='btn13'][@class='btnmar btn2'] \
226         /a[1][@class='abtn2 zabtn']").click()
227     self._d.f("//div[28][@id='btn13'][@class='btnmar btn2'] \
228         /a[1][@class='abtn2 zabtn']").click()
229     self._d.f("//div[28][@id='btn13'][@class='btnmar btn2'] \
230         /a[1][@class='abtn2 zabtn']").click()
231     self._d.f("//div[28][@id='btn13'][@class='btnmar btn2'] \
232         /a[1][@class='abtn2 zabtn']").click()
```