VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

APLIKACE PRO TVORBU SAD TESTŮ GUI

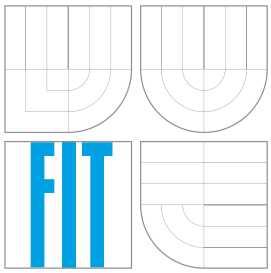
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JURAJ MELO

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

APLIKACE PRO TVORBU SAD TESTŮ GUI

APPLICATION FOR GENERATING GUI TEST SUITE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JURAJ MELO

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ALEŠ SMRČKA, Ph.D.

BRNO 2013

Abstrakt

Tato diplomová práce popisuje systém pro automatizované testování GUI, využívající asistenční technologie pro přístup a manipulaci komponent GUI. Vstupem tohoto systému je popis testovací sady, pro který byl navržen speciální jazyk. Popis testovací sady je tvořen událostmi a aktivitami provedenými v uživatelském rozhraní. Testovací systém automaticky vygeneruje různé sekvence zadaných událostí a aktivit, podle specifikovaného kriteria. Vygenerována testovací sada je pak vykonána interpretem jazyka Python, využívajíc Linux Desktop Testing Project (LDTP). Uvedený systém poskytuje zprávy o průběhu a pokrytí jednotlivých testovacích případů a celé testovací sady.

Abstract

This thesis describes a system for automated GUI testing using assistive technologies for accessing and manipulating GUI elements. The only input from the user to automated test system is a description of UI events and activities. For this purpose, a specialized language is proposed. The test system then automatically generates possible sequences of UI events applying a given criterion. Generated test set is executed by Python interpreter exploiting the Linux Desktop Testing Project (LDTP). Test system described in this thesis then provides reports and coverage evaluation for particular test cases and the whole test set.

Klíčová slova

automatizované testování GUI, LDTP, asistenční technologie

Keywords

automated GUI testing, LDTP, assistive technologies

Citace

Juraj Melo: Aplikace pro tvorbu sad testů GUI, diplomová práce, Brno, FIT VUT v Brně, 2013

Aplikace pro tvorbu sad testů GUI

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Juraj Melo
6. května 2013

Poděkování

Ďekuji mému vedoucímu Ing. Aleši Smrčkovi za jeho pomoc a odborné rady poskytnuté při psaní této práce.

Obsah

Kapitola 1

Úvod

Jednou z kľúčových súčastí aplikácií a softvérových produktov je ich užívateľské rozhranie. Práve s touto časťou softvéru prichádza užívateľ do priameho styku, a preto má veľký vplyv na celkové hodnotenie softvéru. Zásadný vplyv má tiež na použiteľnosť softvéru, keďže je hlavným nástrojom na jeho ovládanie a poskytuje užívateľovi spätnú väzbu.

Človek potreboval komunikovať s počítačom už od samého počiatku ich vývoja. Odvtedy sa vyvinulo niekoľko druhov rozhraní s rôznymi spôsobmi komunikácie. V súčasnosti existuje niekoľko typov užívateľských rozhraní, z ktorých medzi najpoužívanejšie patria: textové užívateľské rozhrania a grafické užívateľské rozhrania. V textových rozhraniach môže užívateľ v jednom okamžiku vykonať iba jednu akciu – príkaz, na ktorú dostane odozvu v podobe výsledku. V grafických rozhraniach je možné vykonať v jednom okamžiku viac udalostí, preto je ich testovanie zložitejšie. Medzi grafické rozhrania patria tiež webové rozhrania, ktoré sú špecifické tým že ovládaná aplikácia beží na inom stroji a je bezstavová.

Grafické užívateľské rozhranie [13], ďalej označované ako GUI, je zložené z objektov nazývaných komponenty (angl. widget). Tieto komponenty využívajú metafory reálneho sveta a patria sem napríklad: tlačidlá, menu, ikony a ďalšie. GUI sú zo svojej podstaty hierarchické, pretože ich jednotlivé komponenty sa združujú do okien, dialógov, prípadne hierarchických menu. Užívateľ s týmito objektmi interaguje podobným spôsobom ako v reálnom svete. Interakcie užívateľa sú v GUI reprezentované udalosťami, ktoré sú následne spracované vykonaním príslušnej obslužnej funkcie. Každá takáto udalosť vyvolá deterministickú zmenu stavu softvéru, ktorá sa môže prejaviť tiež zmenou výzoru niektorých prvkov GUI.

V súčasnosti tvoria GUI jednu z kľúčových častí aplikácií a napriek ich širokému použitiu bolo v minulosti ich testovanie zanedbávané [13]. Podľa [28] je 45 – 60 % zdrojového kódu softvérových aplikácií venovaného práve implementácii funkcionality GUI. Z týchto poznatkov vyplýva, že pri overovaní správnosti systému ako celku je nevyhnutné overiť tiež správnosť funkčnosti GUI. Dôvodom prečo bolo toto testovanie zanedbávané je práve jeho odlišnosť a väčšia zložitosť, ako pri testovaní konvenčného softvéru.

Dnes už existuje množstvo nástrojov, frameworkov a techník testovania GUI, ktoré sa vyvinuli postupom času. GUI je možné testovať rôznymi spôsobmi, pričom každý z nich využíva inú metódu na identifikáciu komponent GUI a manipuláciu s nimi. Jednou z možností ako manipulovať s komponentami testovaného GUI a získavať o nich informácie je využitie asistenčných technológií.

Asistenčné technológie zahŕňajú softvér a hardvér, ktorý poskytuje prístup k informačným technológiám aj ľuďom s rôznym postihnutím. K asistenčným technológiám patrí napríklad program pre čítanie obsahu obrazovky alebo terminál Braillovho písma. Tieto

technológie pridávajú štandardným softvérovým aplikáciam funkcionalitu, ktorá ich robí prístupnými pre čo najširší okruh ľudí.

Cieľom tejto práce je navrhnúť princíp testovania GUI s využitím Linux Desktop Testing Project, navrhnúť jazyk popisujúci testovaciu sadu a metódu automatickej tvorby testovacej sady. Ďalej je cieľom implementovať prekladač tohto jazyka a systém, ktorý otestuje GUI aplikácie na základe vytvorenej testovacej sady, zaznamená výsledok a pokrytie vykonaných testov.

Táto práca čerpá z výsledkov semestrálneho projektu, v rámci ktorého boli spracované teoretické poznatky potrebné k návrhu a implementácií systému pre automatizované testovanie GUI. Tieto poznatky zahŕňajú informácie o testovaní GUI, popis a konkrétne príklady testovania GUI rôznymi metódami, princíp využívania asistenčných technológií a testovanie GUI aplikácií prostredníctvom Linux Desktop Testing Project. Uvedené teoretické poznatky sú spracované v kapitole 2. Súčasťou semestrálneho projektu bolo tiež definovanie podrobných požiadavkov na vytváraný systém uvedených v kapitole 3 a základ návrhu tohto systému a jazyka pre popis testovacej sady.

Návrh systému a jazyka bol v rámci tejto práce upravený a doplnený. Jeho popis je uvedený v kapitole 4. Tento systém bol následne implementovaný, pričom jeho implementačné detaily sú popísané v kapitole 5. Kapitola 6 popisuje experimentálne overenie funkčnosti systému pozostávajúce z otestovania GUI vybraných aplikácií. Výsledky práce sú zhodnotené v kapitole 7.

V prílohe A je uvedený obsah CD nosiča priloženého k tejto práci. Príkazy navrhnutého jazyka pre popis testovacej sady sú popísané v prílohe B. Príloha C obsahuje popisy testovacích sád použité pri testovaní.

Kapitola 2

Teoretický rozbor

Táto kapitola obsahuje teoretické východiská, popis aktuálneho stavu v oblasti testovania GUI a popis používania asistenčných technológií. Poznatky z nej som využil v ďalších kapitolách pri návrhu a implementácií systému pre automatickú tvorbu testovacích sád.

2.1 Princípy testovania GUI

Grafické užívateľské rozhrania patria medzi najpopulárnejšie spôsoby interakcie medzi užívateľom a programom, a preto tvoria kľúčovú časť softvérových systémov a aplikácií. Užívatelia prostredníctvom nich ovládajú softvér tak, že s objektmi GUI manipulujú, čím tvoria udalosti na ktoré softvér reaguje. Môže sa jednať o stlačenie tlačidla, výber položky v menu alebo vpísanie textu. Jednotlivé udalosti bývajú v softvéri ošetrené obslužnými metódami, ktoré môžu pri vykonávaní zmeniť stav programu, prípadne vzhľad niektorých objektov užívateľského rozhrania.

Testovanie funkčnosti GUI [28] je teda nevyhnutnou súčasťou testovania softvéru. Oblasť testovania GUI však bola dlhý čas zanedbávaná, pretože GUI má odlišnú charakteristiku ako konvenčný softvér, a preto je potrebné pristupovať k jeho testovaniu odlišným spôsobom. Testovanie GUI je o niečo zložitejšie ako testovanie konvenčného softvéru a to z nasledujúcich dôvodov.

Prvým z nich je veľké množstvo vstupov, ktoré môžu spôsobiť enormný nárast stavového priestoru aplikácie. Z neho vyplýva obrovské množstvo stavov, ktoré musia byť otestované. V GUI existuje veľký počet možností ako môže užívateľ interagovať s rozhraním, pričom každá interakcia vyvolá udalosť a každá udalosť môže dostať aplikáciu do iného stavu. Každá udalosť by tiež mala byť otestovaná vo všetkých stavoch aplikácie. Taktiež rôzne sekvencie udalostí môžu vyústiť do rôznych stavov aplikácie.

Ďalším problémom je nevhodnosť použitia štandardných kritérií pokrytia testov, ktoré sa používajú pri testovaní konvenčného softvéru. Metriky ktoré sú založené na určení objemu a typu vykonaného zdrojového kódu, nie sú pri testovaní GUI vhodné. Časť zdrojového kódu implementujúcu funkčnosť GUI je nutné otestovať v čo najväčšom počte stavov, v ktorých môže byť vykonaná. Otestovať všetky možné stavy je značne nepraktické, a preto je nutné otestovať vhodnú podmnožinu stavov.

Pri testovaní je potrebné verifikovať stav GUI pri každom kroku vykonávania testovacieho prípadu. Ak sa pri testovaní dostane aplikácia do neočakávaného stavu, môže byť pokračovanie testovania zbytočné, pretože GUI sa môže na základe tohto stavu neočakávane zmeniť. Jedná sa napríklad o zobrazenie chybovej správy pri zlyhaní vykonávanej operácie.

V takomto prípade musí byť vykonávanie testu prerušené hneď ako je detekovaná chyba.

Problémom je taktiež regresné testovanie [1] . Tento typ testovania je problematický, lebo mapovanie vstupov a výstupov GUI a programu nemusí byť rovnaké v po sebe nasledujúcich verziách aplikácie. Okrem toho poloha a grafické znázornenie prvkov GUI môže byť odlišné. Regresné testovanie je dôležité, pretože pri vývoji GUI sa intenzívne používa prototypovanie.

2.2 Spôsoby testovania GUI

V súčasnosti existuje niekoľko metód na testovanie grafických užívateľských rozhraní. Pre každú z týchto metód existujú nástroje, ktoré ju používajú na testovanie. V tejto práci predstavím a zhodnotím niekoľko konkrétnych technológií, z ktorých každá používa iný spôsob práce s GUI. Najskôr to bude *Xnee*, ktorý pracuje s GUI prostredníctvom nahrávania akcií užívateľa do súboru a ich následného prehrávania. Ďalšou technológiou bude *Sikuli*, ktorá využíva grafický popis a identifikovanie komponent GUI. Na záver predstavím technológiu *LDTP – Linux Desktop Testing Project*, ktorá využíva asistenčné technológie a ktorú využijem aj ďalej vo svojej práci.

Okrem spomenutých technológií existujú tiež knižnice [10] , ktoré sú súčasťou komplexných frameworkov pre vývoj aplikácií s GUI. Tie poskytujú funkcionalitu pre emuláciu užívateľských akcií a zisťovanie stavu komponent GUI. Príkladom je *QTestLib*, ktorá je určená pre unit testovanie [2] GUI aplikácií založených na frameworku Qt. Táto knižnica spája typickú funkcionalitu frameworkov pre tvorbu unit testov spolu s funkcionalitou testovania GUI.

2.2.1 Xnee

Podľa [11] je *Xnee* súbor programov, ktoré pracujú v prostredí X11, dokážu nahrávať, prehrávať a distribuovať akcie užívateľa. Tieto programy je vhodné použiť na nasledovné činnosti.

- Automatizáciu testovania – používateľ si nahrá a uloží sériu akcií, ktorými otestuje funkčnosť GUI a následne ich prehráva podľa potreby, napríklad v podobe regresného testovania.
- Demonštrovanie práce s programom – používateľ si nahrá a uloží akcie ktoré chce prezentovať a následne ich pri prezentácií prehrá.
- Distribuovanie akcií – akcie, ktoré užívateľ vykonáva na jednom počítači môžu byť distribuované na viac staníc.
- Vytváranie makier – používateľ prehráva zaznamenané akcie pomocou klávesovej skratky, ktorú nastaví napríklad programom *xrebind* [3] .
- Opisovanie súborov – napríklad pri testovaní textového editoru je možné vpísať obsah do súboru prehratím zaznamenaných akcií a následne porovnať jeho obsah s originálom.

[1] Podľa [8] sa jedná o opätovné otestovanie zmeneného softvéru, ktorým sa overuje že sa danou zmenou nezaniesla do softvéru nová chyba.

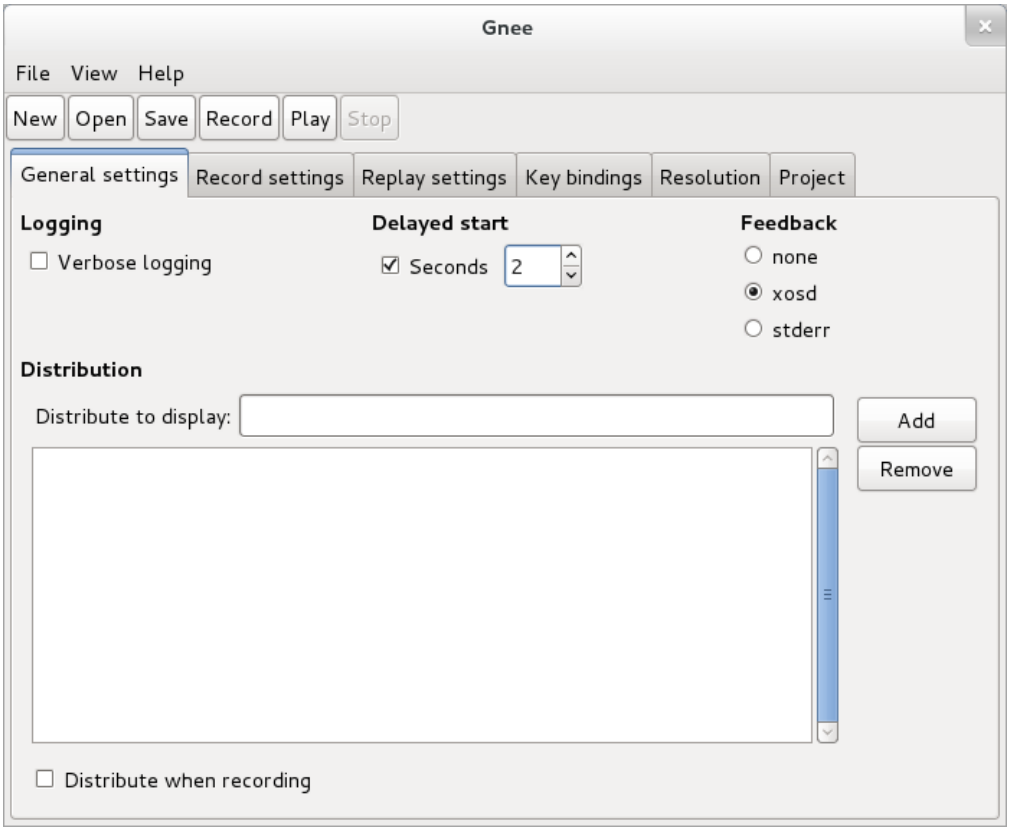
[2] Jedná sa o testovanie na najnižšej úrovni, kde sa testujú jednotlivé metódy, prípadne triedy softvéru. [8]

[3] http://savannah.nongnu.org/projects/xrebind/

Táto skupina nástrojov reprezentuje testovanie typu *record-playback* [12], ktoré bolo často využívané hlavne v minulosti ako jeden zo spôsobov automatizácie testovania GUI. Testovanie prebieha tak, že tester ručne vykonáva akcie v rozhraní testovanej aplikácie, čím generuje udalosti ktoré sú zachytávané nástrojom a následne uložené do súboru spolu s prípadnými snímkami obrazovky. Neskôr môže tester reprodukovať uložené udalosti s inými vstupmi a otestovať tak aplikáciu. Tento spôsob je samozrejme extrémne pracný a jeho úspešnosť závisí na schopnosti testera vybrať vhodné sekvencie udalostí na testovanie.

Podľa [14] obsahuje *Xnee* nasledovné časti, ktoré zabezpečujú uvedenú funkcionalitu:

- *cnee* – program príkazového riadku,
- *gnee* – program ovládateľný prostredníctvom GUI, ktorého náhľad je na obrázku 2.1,
- *pnee* – GNOME applet a
- *libxnee* – knižnicu, ktorú používajú všetky uvedené nástroje.



Obr. 2.1: Grafické užívateľské rozhranie programu *gnee*

Programy *cnee* a *gnee* poskytujú užívateľovi rovnakú funkcionalitu prostredníctvom knižnice *libxnee*, ktorú oba používajú. Jedná sa hlavne o zaznamenávanie udalostí do súboru, prehrávanie zaznamenaných udalostí a distribuovanie udalostí na viac obrazoviek. Program *cnee* má však oproti *gnee* výhodu v tom, že je vhodný na použitie pri automatických regresných testoch. Táto funkčnosť môže byť dosiahnutá napríklad použitím nástroja *cron* [4].

[4] http://linux.die.net/man/8/cron.

Dôležitou vlastnosťou Xnee je, že všetky programy využívajú prostredníctvom knižnice pri svojej práci *X server* [15] . Jedná sa program, ktorý je súčasť systému *X Window*, využíva grafický hardvér a vykonáva samotné vykresľovanie komponent GUI na obrazovku. S týmto serverom komunikujú klienti, v našom prípade programy *cnee*, *gnee* a *pnee*, prostredníctvom X protokolu.

Komunikácia klientov a X servera sa skladá z nasledujúcich typov správ:

- `request` – posiela klient na server, pričom od neho požaduje vykonanie nejakej akcie alebo poslanie dát,
- `reply` – posiela server klientovi ako odpoveď na nejaký požiadavok typu `request`,
- `event` – posiela server klientovi ako informáciu o akcií, ktorú urobil užívateľ a na ktorú by mal klient reagovať a
- `error` – posiela server klientovi v prípade, že od neho obdržal neplatnú správu typu `request`.

Pri každej akcií používateľa pošle *X server* príslušnému klientovi jednu alebo viac správ reprezentujúcich udalosti, ktoré užívateľ vykonal. Tieto udalosti sú výsledok interakcie užívateľa a nazývajú sa *device events*. Môže to byť napríklad stlačenie tlačidla myši, pohyb myšou alebo stlačenie klávesy. Spomenuté udalosti je možné zaznamenať do súboru a následne ich opakovane prehrať. Toto je umožnené rozšíreniami *RECORD*, *XTrap* a *XTest*. *RECORD* dokáže posielať kópiu dát posielaných medzi serverom a klientom klientovi, ktorý tieto dáta požaduje a tým zaznamenať interakciu používateľa s GUI do súboru. *XTest* zase umožňuje reprodukovanie takto zaznamenaných udalostí zo súboru, prípadne ich falšovanie.

Zhodnotenie

Vyskúšal som si prácu s Xnee v operačnom systéme Fedora 17, s jadrom Linuxu verzie 3.4.6 a s prostredím GNOME 3.4.2. Pri testovaní som zistil, že výhodou tohto prístupu je jeho intuitívnosť vytvárania jednoduchých testovacích prípadov.

Na druhej strane vytváranie komplexnejších testovacích sád je veľmi zdĺhavé a náročné. Keďže vytváranie testovacích sád je manuálne, trpí tým ich kvalita, ktorá závisí hlavne na výbere testovaných akcií. Ďalšou prekážkou pri testovaní je nutnosť zachovávať pozíciu testovaných grafických prvkov. V prípade, že má GUI aplikácie pri testovaní inú veľkosť alebo umiestnenie, prehrávané udalosti sa netrafia na požadované komponenty GUI. Testovanie aplikácií je možné vykonávať iba na systémoch používajúcich X server.

2.2.2 Sikuli

Open-source projekt *Sikuli* [20] , vyvíjaný v rámci skupiny *Sikuli Lab*[5], umožňuje automatizovať akúkoľvek činnosť viditeľnú na obrazovke a tým testovať GUI aplikácií. Využíva pri tom systém *Sikuli Search* na vyhľadávanie grafických vzorov na obrazovke a jazyk *Sikuli Script* pre tvorbu vizuálnych testovacích skriptov.

Výhodou použitia technológie Sikuli [9] je možnosť automatizovať každú činnosť viditeľnú na obrazovke počítača bez nutnosti inštalácie dodatočných knižníc. To znamená, že prostredníctvom jazyka Sikuli Script sa dajú ovládať webové stránky a desktopové programy

[5] http://lab.sikuli.org/

obsahujúce GUI bez ohľadu na platformu, na ktorej boli vyvíjané. Pokiaľ je GUI testovaného programu rovnaké, jedným skriptom dokážeme otestovať tento program na rôznych operačných systémoch (Windows, Linux, Mac OS X). Taktiež je možné programovo ovládať a testovať rozhrania aplikácií pre iPhone alebo Android, napríklad prostredníctvom simulátora.

Ďalším prínosom tejto technológie je názornosť testovacích prípadov, z ktorých je možné jednoducho pochopiť aká funkcia je nimi testovaná. Sikuli tiež umožňuje oddeliť vzhľad aplikácie od jej logiky. Testovacie prípady môžu byť vytvorené testermi, ktorým postačuje poznať GUI a následne môžu byť predané programátorom, ktorý si vyvíjané GUI otestujú.

Sikuli v súčastnosti obsahuje niekoľko nedostatkov. Prvým je nemožnosť detekovať neočakávané vizuálne reakcie užívateľského prostredia. Jedná sa o rôzne neočakávané chybové správy alebo dialógy, ktoré sa zobrazia napríklad pri zlyhaní vykonávanej operácie. Taktiež nie je možné detekovať neočakávané chyby vo vykresľovaní grafických komponent GUI. Ďalším nedostatkom je možnosť testovať prostredníctvom Sikuli iba grafické užívateľské rozhranie, ale nie logiku programu. Preto je nutné používať Sikuli spolu s ďalšími testovacími nástrojmi.

Sikuli podporuje dobré testovacie praktiky ako sú unit testovanie a regresné testovanie. Pri unit testovaní sa testovaná aplikácia rozdelí na menšie logické celky, ktoré sa samostatne otestujú. Podpora unit testovania v Sikuli je inšpirovaná frameworkom *JUnit*[6]. To znamená, že tester môže vytvoriť test vo forme funkcie v jazyku Python spolu s príslušnými funkciami na prípravu a vyčistenie testovacieho prostredia. Ďalej môže vytvoriť funkcie, ktoré dostanú GUI do žiadaného stavu (napríklad zobrazia dialóg pre uloženie súboru) alebo otestujú očakávanú vizuálnu odpoveď na vykonanú akciu. Tieto funkcie môžu byť použité viackrát v rôznych testovacích prípadoch. K dispozícií je tiež monitorovanie prebiehajúcich testov a ich vyhodnotenie. Regresné testovanie je podporované možnosťou vytvoriť testovací prípad, ktorý sa dá nad testovaným GUI opakovane spúšťať.

Popis technológie

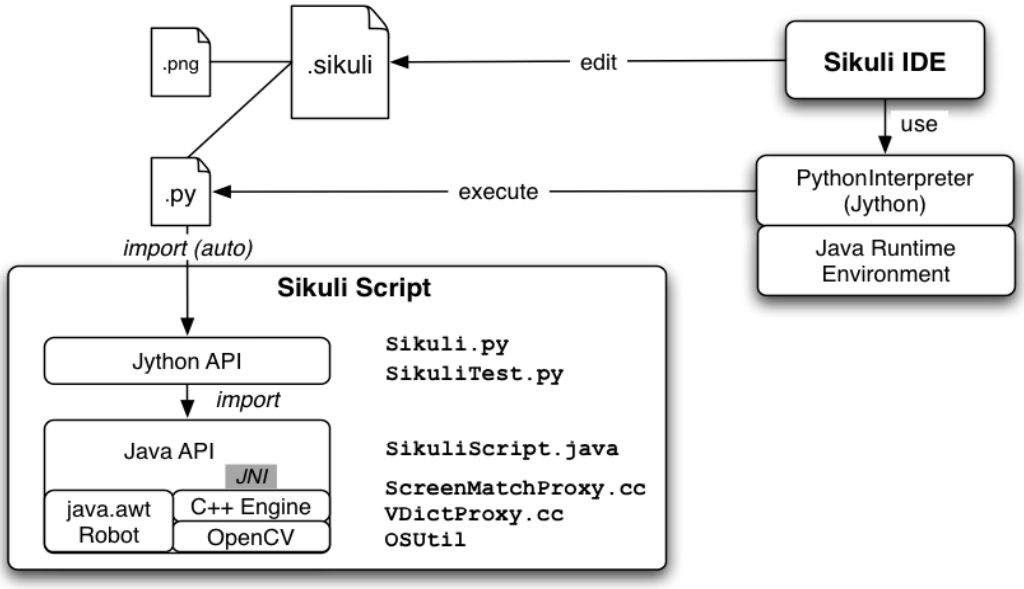
Informácie v tejto časti o technológii Sikuli sú čerpané z [17]. Obrázok 2.2 prehľadne zobrazuje vzťahy medzi jednotlivými časťami technológie Sikuli a ich spoluprácu pri ovládaní alebo testovaní grafického užívateľského rozhrania.

Hlavnou časťou tejto technológie je jazyk *Sikuli Script*. Skladá sa zo samotného jazyka a Java knižnice, ktorou sa automatizuje vyhľadávanie komponent užívateľského rozhrania na obrazovke a interakcia s nimi. Jadrom Sikuli Scriptu je Java knižnica, ktorá obsahuje dve časti. Prvou je `java.awt.Robot`, ktorá doručuje udalosti o práci s myšou alebo klávesnicou do vybraných komponent užívateľského rozhrania, a tým zabezpečuje interakciu používateľa s rozhraním.

Druhou časťou je C++ modul, ktorý pomocou knižnice *OpenCV*[7] vyhľadáva na obrazovke grafické vzory. Tento modul využíva rozhranie JNI – Java Native Interface pre interakciu s Java API a musí byť skompilovaný na každej platforme, na ktorej je nasadený. Grafické vzory vyhľadávané týmto modulom sú komponenty GUI, s ktorými chce užívateľ pracovať. Vzor, ktorý má byť na obrazovke vyhľadaný, je uložený v súbore, vo forme obrázku. Nad knižnicou *OpenCV* sa tiež nachádza vrstva jazyka Jython, ktorá poskytuje užívateľom niekoľko jednoduchých príkazov. K tejto vrstve by malo byť jednoduché pridať niekoľko ďalších vrstiev pre jazyky bežiace na JVM ako sú: JRuby, Scala, Javascript a iné.

[6] JUnit je framework umožňujúci unit testovanie aplikácií v jazyku Java. http://junit.org/

[7] http://opencv.org/



Obr. 2.2: Prehľad práce s technológiou Sikuli. Obrázok bol prevzatý z [17].

Kompletný skript technológie Sikuli je tvorený adresárom s koncovkou `.sikuli`, ktorý obsahuje zdrojový súbor v jazyku Python (`.py`) a všetky obrázky, s ktorými tento súbor pracuje (`.png`). Výhodou je, že každý obrázok je v rámci skriptu reprezentovaný ako cesta k samotnému súboru s obrázkom, a preto je možné upravovať zdrojový súbor aj obyčajným textovým editorom bez nutnosti použitia Sikuli IDE.

Okrem vyššie popísaného skriptu je možné vytvoriť tiež spustiteľný skript (`.skl`), ktorý je tvorený archívom obsahujúcim všetky súbory z adresára tvoriaceho normálny skript. Rozdiel medzi spomenutými typmi skriptov je v spôsobe ich spracovania prostredím Sikuli IDE. Ak predložíme Sikuli IDE ako parameter skript, bude zobrazený v tomto prostredí a je možné ho upravovať. V prípade predloženia spustiteľného skriptu je tento skript okamžite vykonaný bez predchádzajúceho otvárania v Sikuli IDE.

Integrované vývojové prostredie Sikuli IDE zjednodušuje a integruje zachytávanie jednotlivých komponent grafického užívateľského rozhrania a písanie skriptov v špeciálnom textovom editore *SikuliPane*. Tento editor zobrazuje priamo v texte skriptu obrázky komponent grafického rozhrania, s ktorými sa pracuje, a preto sú takto vytvárané skripty mimoriadne intuitívne.

Sikuli Search

Sikuli Search [29] je systém pre vyhľadávanie komponent GUI na obrazovke vo forme grafických vzorov. Vyhľadávanie je implementované hybridnou metódou, ktorá používa rôzne prístupy pre vyhľadávanie vzorov s malými rozmermi a s veľkými rozmermi.

V prípade, že je hľadaný vzor malý (ikona, alebo tlačidlo), používa sa metóda *cross-validation*. Vyhľadávanie sa môže opakovať s rôznym zväčšením alebo zmenšením vzoru, aby sa eliminoval vplyv prípadnej zmeny rozlíšenia obrazovky. Taktiež je možné previesť daný vzor do čiernobielej farby, čím sa pri vyhľadávaní pokryjú rôzne farebné témy GUI testovaného programu.

Ak majú hľadané vzory veľké rozmery (okno, dialóg), vyhľadávanie touto metódou môže byť značne neefektívne a je nutné použiť iný prístup. Vtedy sa používa algoritmus založený na lokálnych invariantných rysoch, ktorý bol pôvodne určený k detekcii vozidiel a ľudí na ulici. Použitý algoritmus najskôr vytvorí objektový model hľadaného vzoru, ktorý je invariantný k veľkosti a natočeniu. Tento model je následne vyhľadaný na obrazovke. Zaujímavosťou je, že toto vyhľadávanie je rotačne invariantné. V súčasných 2D grafických užívateľských rozhraniach táto invariantnosť nie je nutná, no môže byť užitočná pri použití v GUI novej generácie [8].

Sikuli Script

Sikuli Script [29] je jazyk určený na automatizáciu činností s GUI aplikácií. K plnohodnotnému skriptovaciemu jazyku pridáva interaktívne funkcie založené na práci s obrázkami.

Podľa [18] je Sikuli Skript vytvorený ako *Jython* [9] knižnica, a preto je možné používať v ňom syntax jazyka Python. Samotné skripty sa dajú spúšťať prostredníctvom príkazového riadku *Sikuli IDE*, alebo v rámci ďalších integrovaných vývojových prostredí.

Tento jazyk obsahuje niekoľko komponent: funkcia `find()`, triedy `Pattern` a `Region`, a súbor akcií. Funkcia `find()` slúži na lokalizáciu hľadanej komponenty GUI, s ktorou chce užívateľ pracovať. Jej argumentom je vzor, ktorý špecifikuje grafický vzhľad hľadanej komponenty. Funkcia prehľadá obrazovku, prípadne jej časť, a vráti región zodpovedajúci zadanému vzoru. Trieda `Pattern` reprezentuje vyhľadávaný vzor. Môže mať formu obrázku obsahujúceho vzhľad danej komponenty alebo formu textu. Táto trieda ďalej obsahuje niekoľko metód, ktoré určujú ako veľmi špecifická má byť nájdená zhoda. Je možné požadovať presnú zhodu, kedy sa vo výsledku zhoduje každý pixel s hľadaným vzorom, alebo približnú zhodu, ktorej je potrebné zadať prah podobnosti. Ďalej sa dá vyhľadávať nezávisle na farbe a na veľkosti vzoru a výsledku. Trieda `Region` je abstrakciou časti obrazovky, ktorá môže byť použitá ako vstup pri hľadaní komponent (komponenta sa hľadá iba v danej časti obrazovky) alebo ako výsledok vyhľadávania komponent. Okrem toho poskytuje operátory, ktoré umožňujú špecifikovať okolie regiónu: `left`, `right`, `nearby`, `outside` a ďalšie.

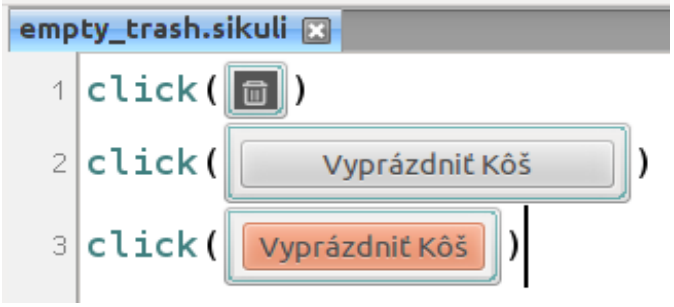
Jazyk tiež definuje akcie, ktoré sa dajú vykonať v strede zadaného regiónu. Sú to napríklad: jednoduché a dvojité kliknutie myšou, vpísanie textu a iné. Nedostatkom jazyka Sikuli Script je nemožnosť pracovať s neviditeľnými prvkami GUI. Jedná sa o prvky, ktoré sú napríklad prekryté iným oknom, alebo sa nachádzajú v aktuálne nezobrazenej záložke.

Rozšírenia Sikuli

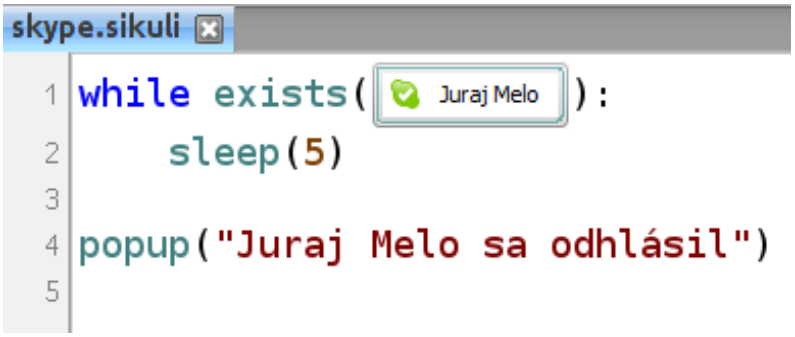
V systéme Sikuli je možné vytvárať novú funkcionalitu vo forme rozšírení [16]. Tieto rozšírenia sú tvorené balíkmi, pričom ich pridávanie je integrované v Sikuli IDE. Príkladom je rozšírenie *Sikuli Guide* [19], ktoré umožňuje jednoduchým spôsobom vytvárať návody (angl. tutorial), pre aplikácie obsahujúce GUI. Hlavným jeho prínosom je možnosť prezentovať pripravené činnosti priamo na skutočnom rozhraní predvádzanej aplikácie. Toto rozšírenie tiež dovoľuje používať všetky možnosti systému Sikuli pre ovládanie GUI a komunikáciu s užívateľom.

[8] Príkladom je rozhranie *tabletop*, v ktorom sú komponenty GUI natočené podľa uhlu pohľadu užívateľa.

[9] Jython je implementácia programovacieho jazyka Python, ktorá umožňuje používať tento jazyk v prostredí *JVM* – Java virtual machine. [26]



Obr. 2.3: Skript na automatické vyprázdnenie koša



Obr. 2.4: Skript na sledovanie aktivity kontaktov v programe Skype

Práca so Sikuli a zhodnotenie

Vyskúšal som si prácu so Sikuli IDE verzie 1.0 RC3 na operačnom systéme Ubuntu 12.04, s jadrom Linuxu verzie 3.2.0 a s použitím prostredia Cinnamon 1.6.3 .

Nevýhodou práce so Sikuli je horšia identifikácia rovnako vyzerajúcich prvkov na obrazovke. V prípade, že máme na obrazovke niekoľko polí na vkladanie textu, musíme použiť ako vzor takú časť obrazovky, ktorou sa dá jednoznačne rozpoznať požadované pole, alebo použiť výrazové prostriedky jazyka Sikuli Script a špecifikovať región, v ktorom sa bude požadované pole vyhľadávať.

Veľkou výhodou však je už spomenutá možnosť testovania a programového ovládania širokého spektra aplikácií bez nutnosti inštalácie špeciálnych knižníc a nástrojov. K výhodám ďalej patrí vytváranie skriptov, ktoré je veľmi intuitívne a ich vzhľad v Sikuli IDE mimoriadne názorný. V neposlednom rade je vhodné použiť práve tento prístup v prípade, že nemáme zdrojové kódy testovanej aplikácie alebo sú testované komponenty GUI inak neprístupné.

Pri študovaní nástroja Sikuli som vyskúšal niekoľko príkladov uvedených v [21] . Prvým príkladom je vyprázdnenie koša. Na obrázku 2.3 je uvedený jednoduchý skript na automatické vyprázdnenie koša. Tento skript pozostáva z troch kliknutí. Najskôr sa otvorí priečinok obsahujúci súbory v koši. Následne sa vyberie akcia na jeho vyprázdnenie a na záver sa akcia potvrdí.

O niečo zaujímavejší je ďalší príklad, ktorým je možné sledovať aktivitu ľudí prihlásených v programe Skype. V skripte na obrázku 2.4 je sledovanie odhlásenia vybranej osoby. Keď jej meno zmizne zo zoznamu prihlásených ľudí, Sikuli o tom informuje používateľa.

2.3 Asistenčné technológie

Podľa [5] môžu byť masovo využívané informačné technológie nedostupné pre ľudí trpiacich postihnutím. Pre slepého človeka je nemožné prijímať informácie poskytnuté aplikáciou vo vizuálnom formáte. Hluchý človek zasa nedokáže pracovať s informáciami v zvukovej forme. Takto vzniknuté bariéry sa dajú odbúrať napríklad tým, že sa pri implementácií softvérových aplikácií použije prístup *univerzálneho dizajnu*.

Univerzálny dizajn [4] je proces vytvárania produktov, ktoré sú dostupné čo najširšej vrstve ľudí, vrátane ľudí s postihnutím. Takto vytvorené aplikácie umožňujú prácu s širokou škálou individuálnych nastavení, efektívne sprostredkúvajú dôležité informácie a sú ovládateľné bez ohľadu na výšku, mobilitu a ďalšie vlastnosti užívateľa.

Aplikácie vytvorené na tomto princípe sa označujú ako *prístupné* (angl. *accessible*) [2]. Aplikácie môžu byť priamo prístupné alebo prístupné s použitím *asistenčných technológií*. Asistenčné technológie [3] sú všeobecne chápané ako technológie, ktoré používajú jednotlivci trpiaci postihnutím, aby mohli vykonávať činnosti, ktoré by inak boli pre nich obtiažne vykonateľné. Pre potreby tejto práce chápeme pod pojmom asistenčné technológie hlavne softvér a hardvér, ktorý postihnutým ľuďom sprístupňuje používanie informačných technológií. Jedná sa napríklad o program na čítanie textu z obrazovky alebo špeciálna klávesnica s nadmerne veľkými tlačidlami.

Okrem softvérových aplikácií existujú snahy sprístupniť postihnutým ľuďom aj obsah webových stránok. Konzorcium W3C vyvinulo dokument *WCAG – Web Content Accessibility Guidelines* [27], ktorý obsahuje pokyny a vysvetlenie ako sprístupniť obsah webu postihnutým ľuďom. Pojem obsah webu označuje informácie zobrazené na webových stránkach alebo vo webových aplikáciach ako sú: texty, obrázky, zvuky a štruktúra informácií. Tento dokument je používaný ako technický štandard, popisujúci mieru prístupnosti obsahu webových stránok a aplikácií. Aktuálna verzia *WCAG 2.0* obsahuje 12 postupov pre zvýšenie prístupnosti informácií prezentovaných v prostredí internetu, ktoré sú rozdelené podľa štyroch princípov: vnímateľnosť, ovládateľnosť, pochopiteľnosť a robustnosť. Ku každému postupu definuje kritéria úspešnosti na troch úrovniach: A, AA, AAA.

Základné asistenčné technológie a prvky prístupnosti [1] sú dnes dostupné v každom operačnom systéme, ktorý tak ponúka aspoň minimálnu úroveň prístupnosti. Jedná sa o možnosť prispôsobenia si klávesnice, obrazovky, vizuálne upozornenia a programy na čítanie a zväčšovanie textu na obrazovke. Operačný systém Windows vyvinul vlastný framework *Microsoft UI Automation*[10] pre používanie asistenčných technológií, ktorý poskytuje programový prístup ku komponentám užívateľského rozhrania aplikácií. Programom asistenčných technológií tak umožňuje získavať informácie o komponentách GUI a manipuláciu s nimi.

2.3.1 Podpora asistenčných technológií v prostredí GNOME

Na platforme Linux sa asistenčné technológie rozvíjajú hlavne v rámci grafického prostredia GNOME[11], v ktorom existuje projekt *GNOME Accessibility Project* [25]. Prostredie GNOME preto obsahuje množstvo asistenčných technológií, ktoré pomáhajú ľuďom s postihnutím a špeciálnymi potrebami. Tieto technológie môžeme rozdeliť do troch základných kategórií podľa typu postihnutia.

[10] http://msdn.microsoft.com/en-us/library/windows/desktop/ee684009%28v=vs.85%29.aspx
[11] http://www.gnome.org/

- **Zrakové postihnutia** – patrí sem nastavenie kontrastu, veľkosti textu, zväčšenie časti obsahu obrazovky a tiež program *Orca*, slúžiaci na reprezentáciu informácií na obrazovke v zvukovej podobe alebo prostredníctvom terminálu Braillovho písma.

- **Sluchové postihnutia** – vizuálne upozornenie, ktoré namiesto zvuku vyvolá bliknutie okna alebo celej obrazovky.

- **Pohybové postihnutia** – zahŕňajú nastavenia rýchlosti pohybu a rozpoznania dvojitého kliknutia myši, simuláciu kliknutia tlačidlom myši pomocou špeciálnych akcií, ovládanie aplikácií klávesnicou bez použitia myši, vypnutie opakovaného stlačenia klávesy pri jej držaní, ignorovanie viacnásobných stlačení jednej klávesy za sebou a ovládanie klávesnice zobrazenej na obrazovke pomocou myši.

Prostredie GNOME využíva *ATK – The Accessibility Toolkit* [24] popisujúci množinu rozhraní pre podporu asistenčných technológií. Ak komponenty GUI implementujú tieto rozhrania, stanú sa prístupnými a môžu byť používané asistenčnými technológiami. Tieto rozhrania sú nezávislé na použitom prostredí a preto ich môžu implementovať rôzne sady komponent ako sú: GTK, Qt, alebo Motif.

V prípade komponent GUI z toolkitu GTK, sa ich implementácia nachádza v module *GAIL – GNOME Accessbility Implementation Library*, ktorý je dynamicky načítaný GTK aplikáciou počas jej behu. Výhodou je, že v tomto prípade majú všetky štandardné komponenty GUI aplikácie zabezpečenú základnú úroveň prístupnosti bez akýchkoľvek ďalších modifikácií aplikácie. Ak modul GAIL nie je načítaný počas behu aplikácie, GTK komponenty majú k dispozícii iba základnú implementáciu, ktorá neposkytuje asistenčným technológiám žiadne informácie.

Dynamické načítanie tejto knižnice v prostredí GNOME závisí na nastavení v konfiguračnom systéme *GConf*. Aby sa táto knižnica načítala, je potrebné nastaviť hodnotu kľúča `"/desktop/gnome/interface/accessibility"` na `"true"`, čím sa povolí podpora asistenčných technológií. Vtedy sa pri volaní funkcie `gnome_program_init` automaticky načítajú požadované knižnice. V GTK+ aplikáciach, ktoré nepoužívajú knižnicu *libgnome* sa podpora asistenčných technológií povoľuje na základe premennej prostredia `GTK_MODULES`, ktorá musí mať hodnotu `gail:atk-bridge`.

V minulosti bolo potrebné, aby si asistenčné technológie museli udržovať komplexný model používaných aplikácií na základe detekovania udalostí operačného systému, používania nepodporovaných aplikačných rozhraní a ďalších neprenosných techník. To spôsobovalo, že podpora asistenčných technológií bola často závislá na konkrétnom operačnom systéme alebo aplikácií.

Na rozdiel od tohto prístupu používa prostredie GNOME rozhranie *AT-SPI – Assistive Technology Service Provider Interface* (často označované aj ako SPI), ktoré je nezávislé na konkrétnom prostredí. Takto je možné poskytnúť asistenčným technológiám, napríklad programom čítajúcim obsah obrazovky, potrebné informácie o aplikáciach prostredníctvom stabilného rozhrania. Tieto informácie sú poskytované počas behu aplikácií, a preto odstraňujú nutnosť uchovávania modelu používaných aplikácií.

Podpora asistenčných technológií je vstavaná do aplikačných toolkitov prostredníctvom príslušného aplikačného rozhrania. Napríklad aplikácie vytvárané v jazyku C používajú zvyčajne ATK a Java aplikácie využívajú Java Accessibility API. Z nich je táto podpora exportovaná príslušným premostením (angl. bridge) na spoločné rozhranie AT-SPI. Na toto rozhranie sú napojené konkrétne aplikácie asistenčných technológií.

Komponenta GUI sa v prostredí GNOME považuje za prístupnú, a teda podporujúcu asistenčné technológie, ak sa jej použitie riadi všeobecnými pravidlami, a ak implementuje ATK rozhrania v súlade s jej úlohou v GUI.

Vstavanú podporu asistenčných technológií môžu aplikácie využívať bez akýchkoľvek ďalších modifikácií. Podmienkou je, aby využívali iba štandardné GTK alebo GTK+ komponenty a nepoužívali ich spôsobom, ktorý by bol v konflikte s touto vstavanou podporou. Komponenty, ktoré sú jednoducho odvodené od štandardných komponent taktiež zdedia ich podporu asistenčných technológií.

Aj napriek vstavanej podpore je často vhodné doplniť a spresniť informácie o jednotlivých komponentách, ktoré sú poskytované asistenčným technológiám. Toto je možné dosiahnuť volaním príslušných metód z ATK priamo v aplikácií. Vtedy bude mať užívateľ presnejší obraz o daných komponentách, ich stave a úlohe v danej aplikácii. [12]

V prípade, že aplikácia používa špeciálne a neštandardné komponenty, je potrebné ručne implementovať príslušné rozhrania ATK, a tým sprístupniť komponenty asistenčným technológiám.

2.4 Linux Desktop Testing Project

Linux Desktop Testing Project [6], ďalej označovaný ako LDTP, je súbor nástrojov pre automatizované testovanie grafických užívateľských rozhraní prostredníctvom asistenčných technológií. Tieto technológie sú využívané na manipuláciu s komponentami GUI testovanej aplikácie a na získanie informácií o ich štruktúre a vlastnostiach. Systém LDTP je možné použiť na viacerých platformách. LDTP bolo pôvodne vyvíjané pre operačný systém Linux, no dnes existujú verzie aj pre ďalšie operačné systémy. Pod operačným systémom Windows je možné použiť verziu Cobra WinLDTP [13], ktorou sa dajú testovať aplikácie vyvíjané v prostrediach .NET, C++ a Java. Pre operačný systém Mac OS X existuje verzia PyATOM [14], ktorá pracuje s knižnicou ATOMac využívajúcou Apple Accessibility API. Použitie LDTP pod systémom Linux umožňuje testovať aplikácie, ktoré sú implementované v prostredí GNOME, KDE a Java Swing.

Systém LDTP [7] používaním asistenčných technológií obmedzuje okruh aplikácií, ktoré je ním možné testovať. Aplikácie totiž musia mať počas testovania implementované príslušné rozhrania pre podporu asistenčných technológií. V opačnom prípade nie je možné aplikáciu týmto spôsobom testovať. Tento prístup má aj svoje pozitíva. Kedže informácie o komponentách GUI sú získavané jednotným rozhraním asistenčných technológií, nie je problém testovať aplikácie vytvorené naprieč rôznymi toolkitmi ako sú: GTK, QT a iné.

Tento systém zachováva koncepty uvedené v *Software Automation Framework Support* [15] a ponúka množstvo príkazov pre testovanie GUI softvérových aplikácií. V nasledovných odrážkach je zhrnutá funkcionalita, ktorú LDTP ponúka.

- Poskytuje príkazy pre manipuláciu s komponentami GUI, ktoré umožňujú vykonať akcie ako: kliknutie, vpísanie textu, vloženie hodnoty, výber položky a iné.
- Poskytuje príkazy, ktoré zisťujú aktuálne informácie a stav komponent GUI ako sú: príkaz na overovanie existencie požadovaného okna alebo komponenty a príkazy

[12] https://developer.gnome.org/accessibility-devel-guide/stable/gad-coding-guidelines.html.en
[13] https://github.com/ldtp/cobra
[14] https://github.com/pyatom/pyatom
[15] http://safsdev.sourceforge.net/Default.htm

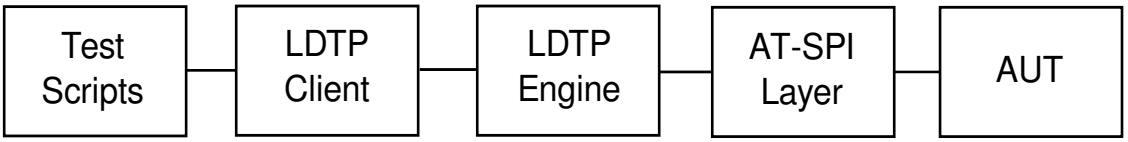
pre výpis atribútov a stavu jednotlivých komponent.

- Poskytuje príkazy na generovanie udalostí simulujúcich prácu s klávesnicou a myšou.
- Umožňuje spracovať neočakávané udalosti v GUI prostredníctvom *callback funkcií*.
- Dokáže monitorovať spotrebu systémových zdrojov testovanej aplikácie.
- Ovládanie tohto systému je možné aj zo vzdialeného počítača.

Systém LDTP umožňuje zaregistrovanie callback funkcií, ktoré budú vykonané v prípade zobrazenia špecifického okna. Týmto spôsobom dokáže detekovať neočakávané udalosti a reagovať na ne. Príkladom neočakávanej udalosti môže byť chyba, ktorá nastane počas testovania aplikácie a spôsobí zobrazenie výstražného okna.

2.4.1 Architektúra

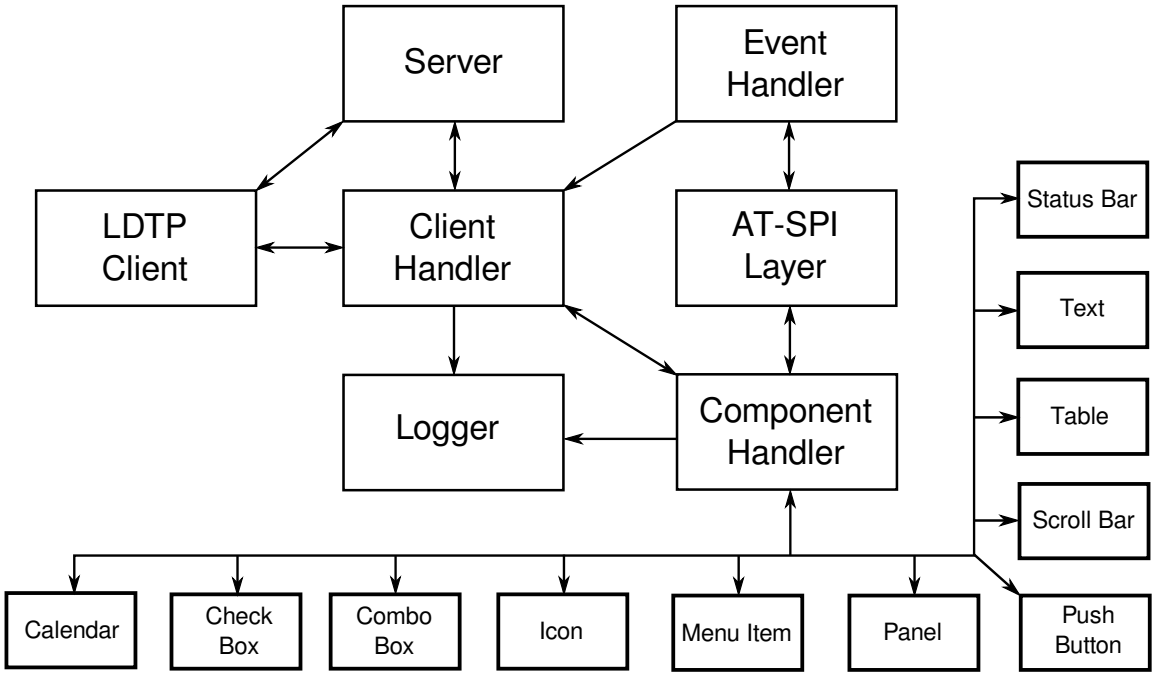
Základná architektúra [7] LDTP je znázornená na obrázku 2.5. Testovacie prípady pre otestovanie GUI danej aplikácie sú popísané vo forme skriptov, ktoré používajú volania LDTP API. Vykonávanie testovacích skriptov je reprezentované LDTP klientom, ktorý môže s modulom *LDTP Engine* komunikovať prostredníctvom UNIX socketov, TCP socketov a protokolom XML RPC. LDTP Engine ďalej využíva knižnicu AT-SPI pre komunikáciu s testovanou aplikáciou (na obrázku 2.5 označenej ako AUT).



Obr. 2.5: Základná architektúra systému LDTP. Obrázok bol prevzatý z [7] a upravený.

LDTP poskytuje množstvo príkazov, ktoré dokážu manipulovať s komponentami GUI podobne ako užívateľ a zisťovať ich stav. Väčšina príkazov vyžaduje zadanie aspoň dvoch argumentov. Tieto argumenty jednoznačne určia objekt (komponentu GUI), na ktorom sa má daná operácia vykonať. Prvým argumentom je identifikácia okna reprezentujúceho kontext a druhým argumentom je identifikácia samotného objektu v danom kontexte. Potom nasledujú ďalšie argumenty potrebné k vykonaniu operácie.

Obrázok 2.6 ukazuje ako pracuje systém LDTP. Na začiatku testovania sa spustí vykonávanie testovacieho skriptu, ktoré začína nadviazaním spojenia medzi LDPT klientom a LDTP serverom. Pri vykonávaní každého príkazu zabalí klient príslušné dáta do formátu XML a pošle ich na server. LDTP Engine tieto dáta na serveri spracuje a na základe ich obsahu spustí príslušný *Component Handler*, ktorý pre komunikáciu s testovanou aplikáciou používa rozhranie AT-SPI. Pre každú komponentu GUI existuje vlastný Component Handler. Po vykonaní príkazu je vygenerovaná odpoveď vo formáte XML, ktorá sa zašle späť klientovi. Táto odpoveď môže obsahovať informáciu o úspechu alebo neúspechu príkazu a pri špeciálnych príkazoch zisťujúcich stav GUI môže odpoveď obsahovať požadovanú hodnotu. Pre spracovanie neočakávaných okien a dialógov je určený *Event Handler*. Aby mohla byť neočakávaná udalosť spracovaná, je nutné zaregistrovať si tzv. *callback funkciu*, ktorá sa v prípade potreby vykoná a obslúži tak vzniknutú udalosť.



Obr. 2.6: Schéma použitia LDTP. Obrázok bol prevzatý z [7] a upravený.

2.4.2 Aplikačná mapa

Aplikačná mapa (angl. *application map*) [7] je v systéme LDTP textovou reprezentáciou GUI testovanej aplikácie. V tejto mape sú podľa definovaných konvencií LDTP uvedené identifikátory všetkých objektov GUI ako sú okná, tlačidlá, položky menu a ďalšie. Počas testovania pristupuje LDTP k týmto komponentám práve s využitím vygenerovanej aplikačnej mapy. Ku komponentám GUI sa v systéme LDTP pristupuje z testovacích skriptov prostredníctvom identifikátora komponenty, ktorý je uvedený v aplikačnej mape.

Komponenty GUI sú v systéme LDTP organizované hierarchicky do okien tak, ako aj v reálnej aplikácií. Každá komponenta obsahuje atribúty ako napríklad: typ komponenty, popisok, rodič alebo potomkovia. Okrem toho môže mať každá komponenta stavy, ktoré reflektujú aktuálny stav zodpovedajúcej komponenty v GUI testovanej aplikácie. Stavy komponent môžu byť: viditeľná, označená, editovateľná, horizontálna a ďalšie.

Pri testovaní potrebujeme programovo pristupovať k dvom základným typom objektov. Sú nimi okno a komponenta nachádzajúca sa v okne. Pre okno platí, že jeho identifikátor je tvorený trojznakovou predponou, ktorá určuje typ okna. Pre okno typu frame je to predpona `frm`, pre okno typu Dialog, Alert a File Chooser je to predpona `dlg`. Všeobecné okná bez titulku majú tiež predponu `dlg`. Za touto predponou nasleduje názov okna, ktorý je tvorený jeho titulkom, z ktorého sú odstránené prípadné znaky medzera a nový riadok. To znamená, že napríklad okno aplikácie *gedit* s titulkom `'Unsaved Document 1 - gedit'`, bude mať výsledný identifikátor `'frmUnsavedDocument1-gedit'`.

V prípade, že okno neobsahuje titulok alebo je v danom momente zobrazených viac okien s rovnakým titulkom, na koniec identifikátoru sa pridá číslo, ktoré ich rozlíši. Pri špecifikovaní okna v testovacom skripte je namiesto identifikátora okna možné uviesť jeho titulok. Okrem toho je samozrejme možné špecifikovať okno priamo jeho identifikátorom. V každom prípade sa však dajú použiť znaky `*` a `?`. Znak `*` reprezentuje nula alebo viac ľubo-

voľných znakov a znak `?` reprezentuje jeden ľubovoľný znak. Nasledujúce reťazce sumarizujú možnosti ako špecifikovať konkrétne okno v testovacom skripte: `'Unsaved Document 1 - gedit'`, `'frmUnsavedDocument1-gedit'`, `'*-gedit*'`, `'dlgAppoinment1'` alebo `'dlg0'`.

Identifikátor komponenty nachádzajúcej sa v okne je tiež tvorený trojznakovou predponou udávajúcou typ komponenty a hodnotou jeho atribútu label alebo associated label. Komponenty ktoré obsahujú popisok (angl. *label*), ako sú napríklad tlačidlá (Push Button) alebo položky menu (Menu Item), používajú text popisku. Komponenty typu Check Box, Radio Button alebo Combo Box zase používajú pridružený popisok (angl. *associated label*). Pri tvorbe identifikátora sa z prípadného popisku alebo pridruženého popisku odstraňujú znaky: medzera, bodka, dvojbodka, podčiarkovník a nový riadok. V prípade, že komponenta nemá žiaden z týchto atribútov, bude identifikovaná priradeným indexom. V testovacom skripte je možné špecifikovať komponentu GUI priamo prostredníctvom jej popisku alebo pridruženého popisku, ak ich komponenta obsahuje. Okrem toho je možné používať identifikátor komponenty z aplikačnej mapy a tiež znaky `*` a `?` rovnako ako pri špecifikácií okna. Špeciálnou možnosťou je špecifikovať komponentu pomocou jej typu a indexu. Index v tomto prípade označuje poradie komponenty daného typu v rámci okna. Reťazec `btn#0` tak špecifikuje prvé tlačidlo v aktuálnom okne. V prípade, že nastane situácia kedy v jednom okne existuje viac komponent s rovnakým popiskom alebo pridruženým popiskom, identifikátor prvej komponenty zostáva nezmenený, no ďalším komponentám sa na koniec identifikátoru pridá index. Prvá komponenta teda môže byť aj naďalej špecifikovaná popiskom, ale ostatné komponenty iba identifikátorom s príslušným indexom. Nasledujúce reťazce sumarizujú možnosti ako v testovacom skripte špecifikovať konkrétnu komponentu GUI v rámci aktuálneho okna: `'File'`, `'mnuFile'`, `'mnu#2'` alebo `'txt0'`.

2.4.3 Zhodnotenie

V tejto práci využívam systém LDTP pre prístup k informáciám a manipuláciu jednotlivých komponent GUI testovanej aplikácie. Počas práce s týmto systémom som konštatoval jeho výhodu v tom, že dokáže testovať aplikácie postavené na rôznych toolkitoch. Medzi jeho nevýhody však patrí už spomínaná nutnosť implementácie rozhrania pre asistenčné technológie.

Výzvu pri práci s týmto systémom tvorí miestami náročné špecifikovanie jednotlivých komponent GUI. Veľký počet týchto komponent je identifikovaných svojim popiskom, ktorý sa môže počas práce s aplikáciou meniť. Na rovnaký problém som narazil aj pri špecifikácií okien. Pri tvorbe testovacích skriptov je preto nutné počítať s týmito zmenami.

Kapitola 3

Špecifikácia požiadavkov

Táto práca má za cieľ navrhnúť, implementovať a experimentálne overiť nástroj na testovanie GUI, ktorý pri práci využíva systém LDTP. Je potrebné navrhnúť skriptovací jazyk, ktorý bude popisovať testovaciu sadu a emulovať interakciu užívateľa s testovanou aplikáciou. Ďalej je potrebné navrhnúť metódu pre automatickú tvorbu testovacej sady, za účelom čo najväčšieho pokrytia funkčnosti prvkov GUI testovanej aplikácie. Následne treba implementovať prekladač navrhnutého jazyka a systém, ktorý vytvorenú testovaciu sadu spustí a otestuje tak GUI danej aplikácie. Na záver je nutné experimentálne overiť vytvorený systém vygenerovaním a spustením testovacej sady na netriviálnej aplikácií, a vyhodnotením pokrytia vykonaných testov.

Cieľom tohto systému nie je komplexne otestovať funkčnosť celej aplikácie. Testovanie je zamerané iba na funkčnosť komponent GUI, a preto je pri komplexnom testovaní potrebné použiť ďalšie testovacie nástroje. Navrhovaný systém umožňuje definovať čiastkové testy GUI, spôsob ich automatického zostavenia do výslednej testovacej sady a spôsob vykonávania výslednej testovacej sady.

Pre splnenie vytýčených cieľov som špecifikoval požiadavky, ktoré vyplývajú priamo zo zadania práce alebo z architektúry navrhnutého systému a umožňujú kooperáciu jednotlivých častí výsledného riešenia. Tieto požiadavky sa dajú rozdeliť do niekoľkých skupín: požiadavky na testovanú aplikáciu, požiadavky na jazyk, požiadavky na prekladač, požiadavky na testovaciu sadu a požiadavky na reprezentáciu výsledkov a pokrytia testov.

3.1 Požiadavky na testovanú aplikáciu

Požiadavky, ktoré musí spĺňať testovaná aplikácia sú:

- obsahuje implementované rozhranie AT-SPI pre využívanie asistenčných technológií,
- má povolené používanie asistenčných technológií v rámci operačného systému a
- je testovaciemu systému predaná vo forme spustiteľného binárneho súboru.

Prvé dve uvedené požiadavky spĺňajú aj aplikácie postavené na prostredí GNOME, v ktorom je povolené používanie asistenčných technológií. Uvedeným testovacím systémom je možné testovať nielen celú aplikáciu, ale tiež jednotlivé moduly, napríklad modul obsahujúci časť jej GUI. Aj v tomto prípade však musí byť testovaný modul dodaný vo forme spustiteľného súboru.

3.2 Požiadavky na jazyk

Táto skupina obsahuje požiadavky na skriptovací jazyk, ktorý bude použitý v skriptoch pre popis automatického generovania testovacích sád, tvoriacich vstup prekladača. Jazyk pre popis testovacej sady musí:

- byť jednoducho pochopiteľný,
- umožňovať špecifikovanie testovanej aplikácie a parametrov, s ktorými má byť spustená,
- umožňovať emuláciu užívateľa a jeho akcií,
- obsahovať prostriedky na získanie informácií o stave komponent testovaného GUI,
- obsahovať prostriedky pre overenie očakávaného stavu GUI, a tiež zložitejšie overenie úspešného vykonania testovanej akcie a
- obsahovať prostriedky pre popis automatickej tvorby testovacej sady.

3.3 Požiadavky na prekladač

Prekladač jazyka bude spracovávať skripty, automaticky generovať a spúšťať testovacie sady, a spracovávať výsledky a pokrytie testov. Požiadavky, ktoré musí spĺňať sú:

- schopnosť parsovať príkazy v navrhnutom jazyku, detekovať prípadné syntaktické a sémantické chyby,
- automatické generovanie testovacej sady na základe zadaného popisu testovacej sady,
- zabezpečenie vykonania vygenerovanej testovacej sady, čím sa otestuje GUI testovanej aplikácie a
- spracovávanie výsledkov a pokrytia vykonávaných testov, a ich ukladanie vo forme XML súborov.

3.4 Požiadavky na testovaciu sadu

Prekladač vygeneruje testovaciu sadu vo forme spustiteľných skriptov, z ktorých každý reprezentuje jeden testovací prípad. V rámci testovania je možné vykonávať vygenerované testovacie prípady viackrát, vždy s rôznymi parametrami testovanej aplikácie a v rôznom prostredí. Testovacia sada musí:

- byť schopná testovať GUI s využitím príkazov knižnice LDTP,
- priebežne zaznamenávať výsledky testovania a
- priebežne sledovať pokrytie vykonávaných testov.

3.5 Požiadavky na reprezentáciu výsledkov a pokrytia testov

Výsledky a pokrytie vykonaných testov sa budú ukladať vo forme XML súborov. Súbor obsahujúci výsledky testovania musí:

- obsahovať informácie o priebehu všetkých vykonávaní testovacích prípadov,
- obsahovať parametre testovanej aplikácie a prostredia v rámci jednotlivých vykonávaní testovacích prípadov a
- musí obsahovať informácie o prípadných chybách, ktoré nastali počas testovania.

Súbor obsahujúci informácie o pokrytí vykonaných testov musí:

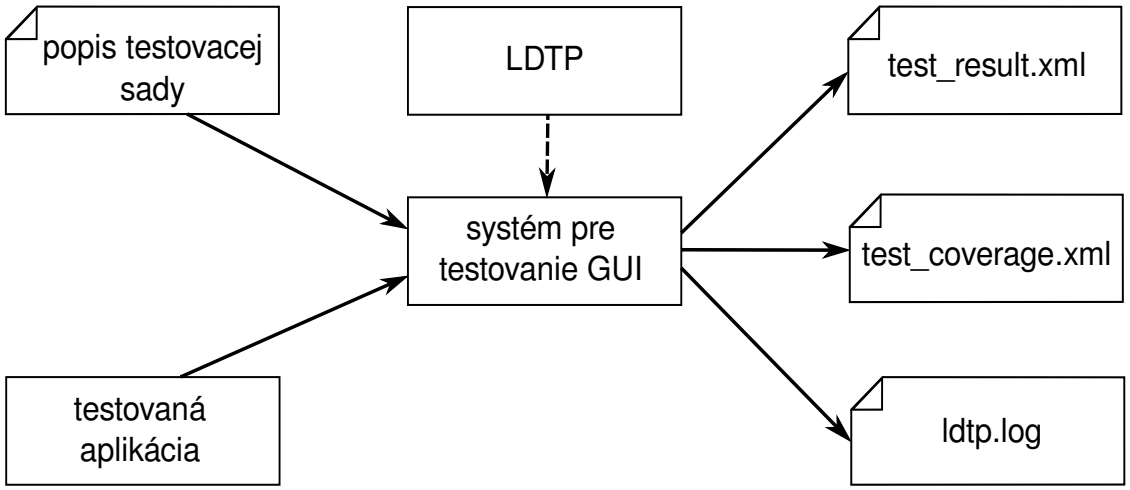
- obsahovať zoznam viditeľných[1] komponent testovaného GUI a okna, v ktorom sa nachádzajú,
- obsahovať informácie o tom, koľkokrát bola daná komponenta GUI pri testovaní použitá,
- obsahovať zoznam aktivít uvedených v popise testovacej sady a informáciu o tom, koľkokrát bola daná aktivita vykonaná a
- uvedené informácie musia popisovať pokrytie v rámci jednotlivých vykonaní testovacích prípadov, ale aj v rámci celej testovacej sady.

[1] Jedná sa o komponenty GUI, ktoré systém LDTP dokáže identifikovať.

Kapitola 4

Systém pre automatizované testovanie GUI

Podľa požiadavkov špecifikovaných v kapitole 3 som navrhol a implementoval systém pre automatizované testovanie GUI. Spôsob použitia tohto systému, jeho vstupy a výstupy sú uvedené na obrázku 4.1 .



Obr. 4.1: Systém pre automatizované testovanie GUI

Uvedený systém pre automatizované testovanie GUI potrebuje pre svoju činnosť systém LDTP a vstupy dodané užívateľom. Vstupy systému sú tvorené popisom testovacej sady a testovanou aplikáciou. Popis testovacej sady je vytvorený v špeciálnom jazyku a popisuje priebeh testovania príslušného GUI. Ďalším vstupom je samotná aplikácia, ktorej GUI bude systém testovať.

Činnosť systému prebieha tak, že na základe dodaného popisu testovacej sady otestuje GUI zadanej aplikácie. Pri tomto testovaní využíva systém LDTP, prostredníctvom ktorého sú sprístupnené komponenty testovaného GUI. Výstupom testovania sú údaje o priebehu, výsledkoch a pokrytí vykonaných testov, a tiež záznam správ systému LDTP.

Táto kapitola popisuje systém nasledovne. Najskôr je v časti 4.1 predstavený jazyk pre popis testovacej sady. Nasleduje popis štruktúry vstupov, výstupov a jednotlivých modulov systému, ktorý je uvedený v časti 4.2 . Činnosť systému a jeho častí je popísaná v 4.3 . Detailný popis spôsobu automatického generovania testovacej sady je uvedený v časti 4.4 .

Na záver sú v časti 4.5 navrhnuté a popísané kritéria pre vyhodnotenie pokrytia vykonaných testov.

4.1 Jazyk pre popis testovacej sady

V rámci tejto práce som navrhol špeciálny jazyk pre popis testovacej sady. Tento jazyk obsahuje upravenú podmnožinu príkazov systému LDTP pre manipuláciu a získavanie informácií z testovaných komponentov GUI. Z dôvodu väčšej názornosti uvádzam aj popis častí jeho syntaxe v Backus-Naurovej forme (BNF).

Príkazy tohto jazyka je možné rozdeliť na tri typy:

1. **Udalosti** – umožňujú emulovať akcie užívateľa a definovať podmienky úspešnosti jednotlivých testovacích prípadov.
2. **Operátory** – slúžia na získavanie informácií o stave jednotlivých komponent GUI. Je z nich možné vytvárať výrazy a testovať ich výslednú pravdivostnú hodnotu.
3. **Riadiace príkazy** – umožňujú špecifikovať testovanú aplikáciu, jej parametre a hodnoty premenných prostredia.

Uvedený jazyk pracuje s hodnotami troch základných dátových typov, ktoré používa aj systém LDTP. Prvým je typ *bool*, ktorý nadobúda hodnoty `true` a `false`. Ďalej typ *int*, ktorý reprezentuje celé čísla a nakoniec typ *string* reprezentujúci textové reťazce. Hodnoty typu string sú v jazyku ohraničené úvodzovkami. Znaky `'` a `"` je možné v týchto reťazcoch uviesť pomocou *escape sekvencie*. Tá je tvorená nasledovne: `\'` a `\"` [1]. V jazyku je taktiež možné používať identifikátory, ktorých syntax je rovnaká ako v jazyku Python. Keďže systém LDTP nepodporuje prácu s textovými reťazcami obsahujúcimi znaky s diakritikou, neodporúčam tieto znaky používať ani v tomto jazyku.

```
<bool> ::= "true" | "false"
<int> ::= celé_číslo
<int-not-negative> ::= nezáporné_celé_číslo
<string> ::= '"'reťazec_znakov'"'
<identifier> ::= reťazec_alfanumerických_znakov_začínajúci_písmenom_
                 alebo_podčiarkovníkom
```

Pri vytváraní skriptov je možné používať komentáre rovnako ako v jazyku C. To znamená riadkový komentár tvorený terminálom `//` a znakmi, ktoré nasledujú za ním až do konca riadku. Okrem neho je podporovaný aj blokový komentár `/* */`, ktorý je tvorený obsahom medzi uvedenými znakmi, a môže obsahovať aj znak nového riadku. Obsah v komentároch sa pri spracovávaní súboru neberie do úvahy.

```
// komentár do konca riadku
/* blokový komentár
   na viac riadkov */
```

[1] Tieto reťazce budú vo výslednej testovacej sade súčasťou skriptov v jazyku Python, a preto je nutné použiť escape sekvenciu aj pre znak `'`.

4.1.1 Aktivity

Popis testovacej sady sa skladá z množiny *aktivít*, v ktorých sú zoskupené udalosti. Každá aktivita musí mať svoj jedinečný identifikátor a telo. Zoskupovanie udalostí do aktivít je prostriedkom dekompozície testovacieho skriptu, ktorý zvyšuje prehľadnosť a spája spolu súvisiace udalosti.

Skript musí obsahovať práve jednu aktivitu, ktorá má identifikátor `main`. Telo aktivity `main` začína neprázdnym zoznamom riadiacich príkazov, po ktorom nasleduje neprázdny zoznam udalostí. Riadiace príkazy môžu byť uvedené vo forme zoznamu príkazov alebo v rámci príkazu `generate-set` s kritériom `random-data-criterion`, ktorý vo svojom tele obsahuje neprázdny zoznam riadiacich príkazov. Aktivity ktoré nemajú identifikátor `main`, majú svoje telo tvorené iba neprázdnym zoznamom udalostí.

Testovanie začína vykonávaním aktivity `main`. Príkazom `do` sa v danom mieste zaháji vykonávanie ďalšej aktivity, ktorej identifikátor je v príkaze uvedený. V systéme sa nekontroluje vznik prípadných cyklov pri volaní aktivít, ani nedostupnosť volania aktivít z aktivity `main`. Toto si pri vytváraní skriptov musí kontrolovať sám užívateľ.

```
<events> ::= <event> |
             <events> <event>
<commands> ::= <command> |
               <commands> <command>
<limit> ::= <int> | "*"
<activity-body> ::=
   <events> |
   <commands> <events> |
   "generate-set" "random-data-criterion" <limit>
      "{" <commands> "}" <events>
<activity> ::= "activity" <identifier> "{" <activity-body> "}"
<file> ::= <activity> |
           <activity> <file>
```

4.1.2 Emulácia akcií užívateľa

V rámci jazyka sú definované udalosti, ktorými je možné emulovať akcie užívateľa vykonané na jednotlivých komponentách testovaného GUI. Väčšinou sa jedná o zapuzdrenie niektorých príkazov dostupných v systéme LDTP. Medzi tieto udalosti patria jednoduché akcie, ako napríklad kliknutie na objekt alebo vpísanie textu.

Okrem nich je možné definovať *blokové udalosti*, ktoré vo svojom tele obsahujú množinu ďalších udalostí. Jedná sa o príkaz `generate-set`, pomocou ktorého sa popisuje generovanie sekvencií udalostí. Ďalej k nim patrí príkaz `if` zložený z výrazu a množiny udalostí, ktoré sú vykonané ak má výraz po vyhodnotení hodnotu `true`.

Udalosti pracujú s jednotlivými komponentami GUI, ktoré sú zoskupené podľa jednotlivých okien aplikácie. Z tohto dôvodu pracujú udalosti vždy v kontexte aktuálneho okna, ktoré sa nastavuje príkazom `on`. Vždy pri nastavení nového aktuálneho okna sa tiež overí, či dané okno existuje.

Po vykonaní akcie je často potrebné pozastaviť testovanie do doby, kým sa v systéme LDTP aktualizujú informácie o GUI aplikácie na základe vykonanej akcie. Toto pozastavenie je možné vykonať príkazom `wait`, ktorý preruší vykonávanie testu na počet zadaných sekúnd. Podrobný popis jednotlivých udalostí je uvedený v prílohe B.1 .

```
<event> ::= "assert" <expression> <string> |
            "click" <string> |
            "do" <identifier> |
            "double-click" <string> |
            <string> "double-click-row" <string> |
            "embedded-test" ľubovoľný_text "embedded-test-end" |
            "generate-set" "event-criterion" "{" <events> "}" |
            "generate-set" "sequence-criterion" <limit>
               "{" <events> "}" |
            "grab-focus" <string> |
            "check" <string> |
            "if" <expression> "{" <events> "}" |
            "keyboard-event" <string> |
            "on" <string> |
            <string> "select" <string> |
            <string> "select-row-contains" <string> |
            <string> "select-tab" <string> |
            <string> "set-text-value" <string> |
            <string> "set-value" <int> |
            "uncheck" <string> |
            "unselect-text" <string> |
            "wait" <int-not-negative> |
            "window-close" <string>
```

Jazyk obsahuje tiež prostriedky pre popis generovania sekvencií udalostí a riadiacich príkazov. Využíva sa na to príkaz `generate-set`, obsahujúci kritérium generovania a množinu udalostí, ktorých sekvencie majú byť podľa neho generované. Existujú tri kritériá generovania sekvencií: `event-criterion`, `sequence-criterion` a `random-data-criterion`. Posledné z nich je vyhradené pre pre generovanie sekvencií riadiacich príkazov a ostatné pre generovanie sekvencií udalostí. Posledné dve uvedené kritériá obsahujú aj svoj *limit* daný hodnotou typu int alebo znakom `*`. Spôsob generovania sekvencií udalostí a riadiacich príkazov je podrobne popísaný v časti 4.4 .

4.1.3 Riadiace príkazy

Aplikácia ktorej GUI bude testované je špecifikovaná *riadiacimi príkazmi*. Tie okrem testovanej aplikácie umožňujú nastaviť aj jej parametre a premenné prostredia, v ktorom bude testovaná. V rámci týchto príkazov je možné definovať premenné typu zoznam. Premenná je tvorená svojim menom a neprázdnym zoznamom hodnôt. Meno premennej má formu identifikátoru a hodnoty uvedené v zozname sú typu string. V riadiacich príkazoch je potom možné namiesto priamej hodnoty typu string uviesť identifikátor premennej. Počas testovania bude daný príkaz získavať hodnoty práve zo zoznamu hodnôt príslušnej premennej.

V jazyku sú definované tri riadiace príkazy. Príkazom `var` je možné definovať vyššie popísané premenné. Ďalej je k dispozícii príkaz `set-enviroment`, ktorý umožňuje nastaviť hodnotu zadanej premennej prostredia (angl. *enviroment variable*), v ktorom sa aplikácia spustí. Hodnota premennej prostredia môže byť v tomto jazyku zadaná pomocou už definovanej premennej. Príkaz `run` slúži na spustenie testovanej aplikácie. Jeho parametrom je textový reťazec, ktorý obsahuje cestu k spustiteľnému súboru testovanej aplikácie a jej parametre. Ako prvá je zadaná cesta k spustiteľnému súboru, ktorá nesmie obsahovať znak medzera. Tento znak totiž oddeľuje cestu k súboru a jednotlivé parametre. Za cestou potom

môžu nasledovať parametre aplikácie, s ktorými bude spustená. Cesta k binárnemu súboru aplikácie a jej parametre sa zadávajú podobne ako pri jej spúšťaní v príkazovom riadku. V tomto prípade však nie je možné používať nástroje príkazového riadku, ako napríklad presmerovanie štandardného výstupu aplikácie. V rámci príkazu `run` musí byť uvedená aspoň cesta k spustiteľnému súboru s testovanou aplikáciou a jej povinné parametre.

```
<list> ::= <string> |
           <string> ”,” <list>
<string-or-identifier> ::= <string> |
                           <identifier>
<command> ::= ”var” <identifier> ”=” ”{” <list> ”}” |
              ”set-enviroment” <string> ”=” <string-or-identifier> |
              ”run” <string-or-identifier>
```

Riadiace príkazy uvedené v zozname musia byť usporiadané podľa definovanej postupnosti. Zoznam začína ľubovoľným počtom príkazov `var`, ktoré definujú premenné. Ďalej nasleduje ľubovoľný počet príkazov `set-enviroment`, ktoré môžu používať definované premenné, a musí byť ukončený práve jedným príkazom `run`. Príkazy `var` a `set-enviroment` môžu byť vynechané, ale všetky premenné použité v zozname riadiacich príkazov musia byť definované. Toto opatrenie pomáha eliminovať používanie nedefinovaných premenných.

4.1.4 Operátory

Informácie o testovaných komponentách GUI a ich stav je možné získať prostredníctvom definovaných operátorov. Tieto operátory definujú operácie, ktoré majú vstupné parametre a vracajú výsledok v podobe hodnoty typu string alebo int. Príkladom takýchto operácií je získanie informácie o tom, či objekt existuje, či je viditeľný, prípadne akú textovú hodnotu obsahuje. Podrobný popis jednotlivých operátorov je uvedený v prílohe B.2 .

```
<operation> ::= ”default” <string> |
                ”editable” <string> |
                ”enabled” <string> |
                ”get-label” <string> |
                ”get-row-count” <string> |
                ”get-tab-count” <string> |
                ”get-text-value” <string> |
                ”checked” <string> |
                ”object-exists” <string> |
                <string> ”row-exists” <string> |
                <string> ”verify-select” <string> |
                ”visible” <string>
```

Príkazy, ktoré vyhodnocujú správnosť takto zistených informácií o testovanom GUI sú: `assert` a `embedded-test`. Príkaz `assert` očakáva dva parametre: výraz a textový reťazec typu string. Počas testovania sa najskôr vyhodnotí uvedený výraz, a ak má výslednú hodnotu `true`, pokračuje sa ďalej v testovaní. V opačnom prípade to znamená, že sme v GUI testovanej aplikácie odhalili chybu. Druhým parametrom je správa popisujúca funkcionalitu otestovanú daným výrazom. Táto správa bude uvedená aj v súbore obsahujúcom priebeh a výsledky testovania. Z dôvodu väčšej prehľadnosti odporúčam zapísať výraz do zátvoriek.

Pre zložitejšie testovanie funkčnosti je určený príkaz `embedded-test`. Jeho telo tvorí reťazec, ktorý je počas testovania interpretovaný ako program zapísaný v jazyku Python (riadky 6 – 9), testujúci požadovanú funkčnosť. V rámci tohto skriptu je pripravený príkaz `my_assert` (riadok 9), ktorý má rovnaké parametre a sémantiku ako príkaz `assert`, používaný v skripte pre popis testovacej sady. Keďže odsadenie príkazov jazyka Python má svoju danú sémantiku, musí byť otvárací terminál `embedded-test` uvedený na začiatku nového riadku. Telo tohto príkazu potom začína na ďalšom riadku a odsadenie má rovnaký význam ako v jazyku Python. Odporúčam zapisovať tento príkaz nasledujúcim spôsobom:

```
1  activity test
2  {
3     on "*-gedit*"
4
5  embedded-test
6  print 'test zacina'
7  if True:
8     print 'telo if'
9  my_assert(True, 'test uspesny')
10 embedded-test-end
11
12    click "mnuQuit"
13 }
```

Obsah príkazu `embedded-test` je pridaný na príslušné miesto vo vygenerovanom testovacom skripte a od okolitého kódu nie je oddelený. Ak je v popise testovacej sady tento príkaz použitý viac krát, môže sa stať že obsah týchto navonok oddelených skriptov bude pri testovaní súčasťou jedného testovacieho skriptu. Preto je vhodné voliť unikátne hodnoty použitých identifikátorov a obmedziť sa na vytváranie jednoduchých čiastkových testov.

```
<expression> ::= <bool> | <int> | <string> |
     <expression> "==" <expression> | <expression> "!=" <expression> |
     <expression> "and" <expression> | <expression> "or" <expression> |
     "not" <expression> | "(" <expression> ")" | <operation>
```

V jazyku sú ďalej definované logické operátory, ktoré umožňujú vytvárať komplexnejšie výrazy a testy. Všetky uvedené operátory majú rovnakú prioritu a vyhodnocujú sa smerom zľava doprava. Pre väčšiu prehľadnosť však odporúčam pridávať do zložitých výrazov zátvorky. Pretypovávanie hodnôt výsledkov operácií na typ bool, požadovaný logickými operátormi, sa riadi pravidlami jazyka Python.

4.1.5 Sémantické obmedzenia popisu testovacej sady

Na záver popisu tohto jazyku uvádzam zhrnutie sémantických pravidiel, ktoré sú kontrolované po spracovaní súbora s popisom testovacej sady. Popis testovacej sady musí spĺňať nasledovné podmienky:

- popis musí obsahovať aspoň jednu aktivitu s identifikátorom `main`,
- práve jedna aktivita v popise musí mať identifikátor `main`,
- identifikátory aktivít musia byť jedinečné v rámci celého popisu,

- príkaz `do` nesmie volať neexistujúcu aktivitu,
- v aktivite môže byť najviac jeden príkaz `generate-set`,
- v tele príkazu `generate-set` sa nesmie nachádzať príkaz `if` ani `generate-set`,
- príkaz `generate-set` sa nesmie nachádzať v tele príkazu `if` ani `generate-set`,
- zoznam riadiacich príkazov sa môže nachádzať iba v aktivite s identifikátorom `main`,
- v zozname riadiacich príkazov sa môžu nachádzať príkazy v nasledujúcom poradí: ľubovoľný počet príkazov `var`, ľubovoľný počet príkazov `set-enviroment` a práve jeden príkaz `run`,
- všetky premenné používané v zozname riadiacich príkazov musia byť definované, a to najviac jeden krát,
- `random-data-criterion` musí mať hodnotu limitu v uzatvorenom intervale <1;100>,
- `sequence-criterion` musí mať hodnotu limitu väčšiu ako 0, a zároveň táto hodnota nesmie byť väčšia ako počet udalostí v tele príslušného príkazu `generate-set` a
- v príkaze `set-enviroment` nesmie byť názov premennej prostredia špecifikovaný prázdnym reťazcom.

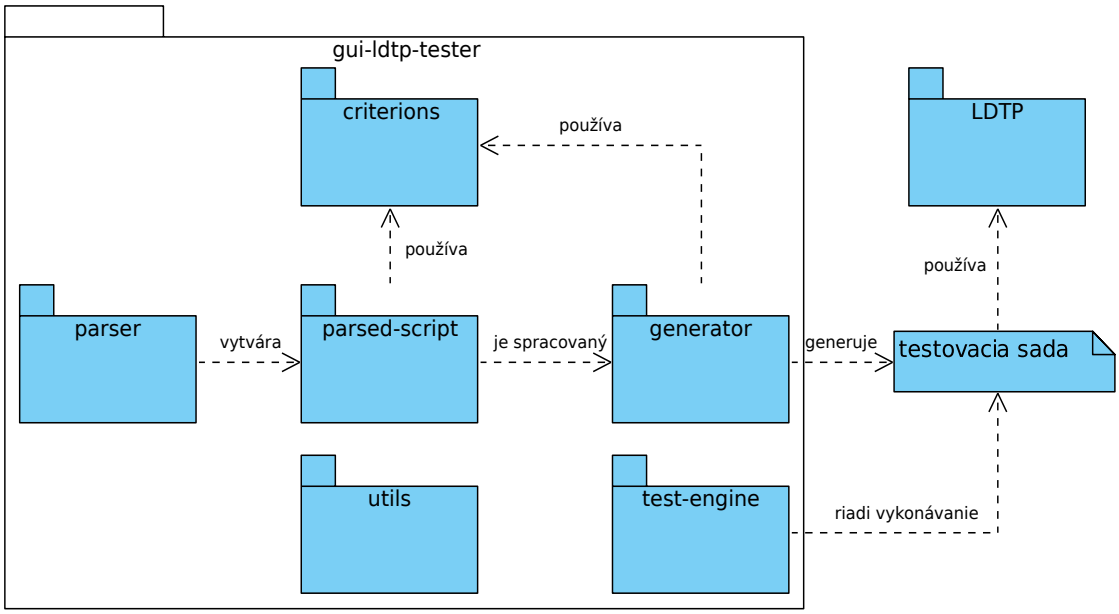
4.2 Štruktúra systému

Systém pre automatizované testovanie GUI pozostáva z programu `gui-ldtp-tester` a testovacej sady, ktorá je týmto programom vygenerovaná. Samotný program `gui-ldtp-tester` sa ďalej skladá z niekoľkých ďalších modulov. Rozdelenie systému na jednotlivé moduly je uvedené na obrázku 4.2 .

Modul *parser* spracováva skript obsahujúci popis testovacej sady. Na základe obsahu tohto skriptu vytvorí jeho internú reprezentáciu. Tá je zložená z objektov tried modulu *parsed-script* vo forme usporiadaného stromu. Jednotlivé triedy z tohto modulu využívajú modul *criterions*, ktorý implementuje generovanie sekvencií udalostí podľa zadaného kritéria. Modul *generator* riadi generovanie testovacej sady z internej reprezentácie poskytnutého popisu testovacej sady. Výsledkom je *testovacia sada*, ktorá je tvorená množinou skriptov testujúcich GUI danej aplikácie. Tieto testovacie skripty využívajú systém LDTP, ktorý je reprezentovaný modulom *LDTP*. Vykonávanie testovacích skriptov a vyhodnocovanie testovania riadi modul *test-engine*. Všetky moduly využívajú modul *utils*, ktorý poskytuje implementáciu často používaných funkcií. Jedná sa hlavne o rôzne konverzie medzi použitými dátovými typmi, spracovanie parametrov príkazového riadku a generovanie častí obsahu skriptov výslednej testovacej sady.

4.2.1 Vstupy systému

Vstupy systému sú tvorené dvomi súbormi. Textový súbor s koncovkou `.tsd` obsahuje popis testovacej sady v jazyku pre popis testovacej sady. Druhým vstupom je spustiteľný súbor testovanej aplikácie. Ten musí po spustení zobraziť testované GUI a dostať aplikáciu do stavu, v ktorom je možné s ňou pracovať. Tento súbor je špecifikovaný v rámci popisu testovacej sady.



Obr. 4.2: Štruktúra systému pre automatizované testovanie GUI

Program `gui-ldtp-tester` očakáva pri spustení povinný parameter, určujúci súbor s popisom testovacej sady. Ďalej je možné zadať nepovinné parametre, ktoré špecifikujú priečinok pre uloženie vygenerovanej testovacej sady a priečinok pre uloženie výstupov testovania.

4.2.2 Štruktúra modulov systému

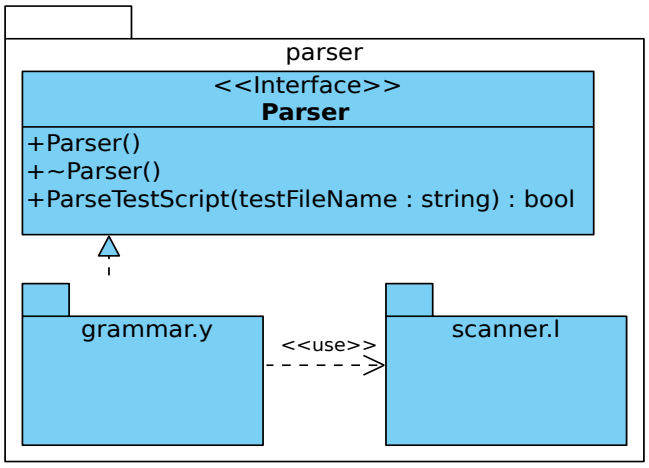
V tejto časti podrobne popisujem štruktúru jednotlivých modulov systému, ktoré boli predstavené vyššie. K týmto modulom uvádzam aj ich diagramy tried jazyka UML. Pre väčšiu prehľadnosť sú v týchto diagramoch uvedené iba významné triedy, ich atribúty a metódy. Nie je v nich modelovaná vedľajšia funkcionalita, ako napríklad štandardné spracovanie parametrov príkazového riadku alebo podpora kontrolných výpisov.

Modul parser

Modul parser, ktorého diagram tried je na obrázku 4.3, spracováva skript obsahujúci popis testovacej sady. Funkcionalita tohto modulu je zapuzdrená v triede `Parser`, ktorá tvorí jeho rozhranie. Spracovanie vstupného súboru pozostáva z kontroly syntaxe a z jeho parsovania podľa syntaxe jazyka pre popis testovacej sady. Táto funkcionalita je implementovaná v popise syntaktického analyzátora `grammar.y`, ktorý pri svojej práci využíva lexikálny analyzátor. Popis lexikálneho analyzátoru je v module `scanner.l`. Výsledkom spracovania popisu testovacej sady je jej interná reprezentácia. Tá je zložená z objektov tried modulu parsed-script vo forme usporiadaného stromu.

Modul parsed-script

V module parsed-script, ktorý je zobrazený v diagrame 4.4, sú obsiahnuté triedy reprezentujúce jednotlivé výrazové prostriedky jazyka pre popis testovacej sady. Operátory sú implementované v triede `Operation`, riadiace príkazy v podmodule *commands* a udalosti



Obr. 4.3: Diagram tried modulu parser

v podmodule *events*. Pre väčšiu prehľadnosť je diagram tried modulu commands uvedený na obrázku 4.5 a diagram tried modulu events je na obrázku 4.6 .

Riadiace príkazy sú v tomto systéme reprezentované objektmi tried modulu commands, ktorý je znázornený v diagrame 4.5 . Základom je abstraktná trieda `Command`, z ktorej sú odvodené triedy všetkých riadiacich príkazov jazyka pre popis testovacej sady. Tento modul obsahuje aj implicitný riadiaci príkaz `Cmd_ExecuteTest`, ktorý nie je súčasťou jazyka a je určený pre samotné vykonanie testovacieho prípadu nad spustenou aplikáciou. Trieda `CommandsList` zapuzdruje prácu so zoznamom riadiacich príkazov.

Modul events obsahuje abstraktnú triedu `Event`, ktorá definuje rozhranie pre preklad jednotlivých udalostí do výsledných testovacích skriptov. Z nej sú odvodené triedy jednotlivých udalostí a blokových udalostí, ktoré toto rozhranie implementujú podľa svojich špecifík. Blokové udalosti sú odvodené z abstraktnej triedy `EventBlock`. Okrem toho je v týchto triedach implementovaná aj funkcionalita sémantickej validácie a tiež generovanie sekvencií udalostí, pri ktorom sa využíva modul criterions. Diagram tried tohto modulu je uvedený na obrázku 4.6 . Z dôvodu väčšej prehľadnosti neobsahuje triedy všetkých udalostí, odvodených z abstraktnej triedy `Event`.

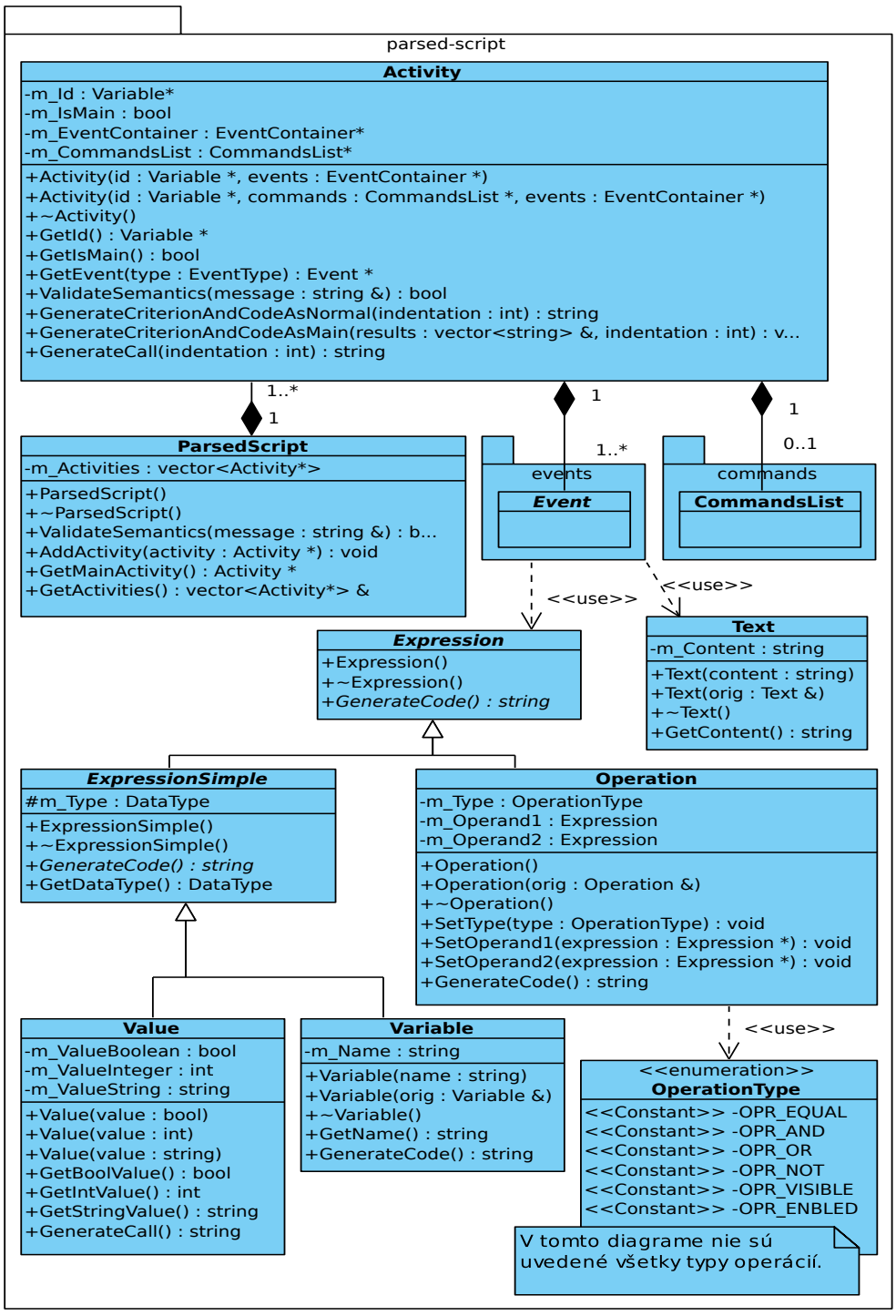
Modul criterions

Diagram tried modulu criterions je uvedený na obrázku 4.7 . Súčasťou tohto modulu je trieda `CriterionGenerator`, ktorá implementuje generovanie sekvencií udalostí a trieda `CommandsCriterionGenerator` implementujúca generovanie výsledného zoznamu riadiacich príkazov. Tieto triedy sú navrhnuté tak, aby ich objekty bolo možné použiť nezávisle pri generovaní jednotlivých sekvencií. Trieda `CriterionResult` slúži na jednotnú reprezentáciu výsledkov generovania sekvencií udalostí a na prácu s nimi.

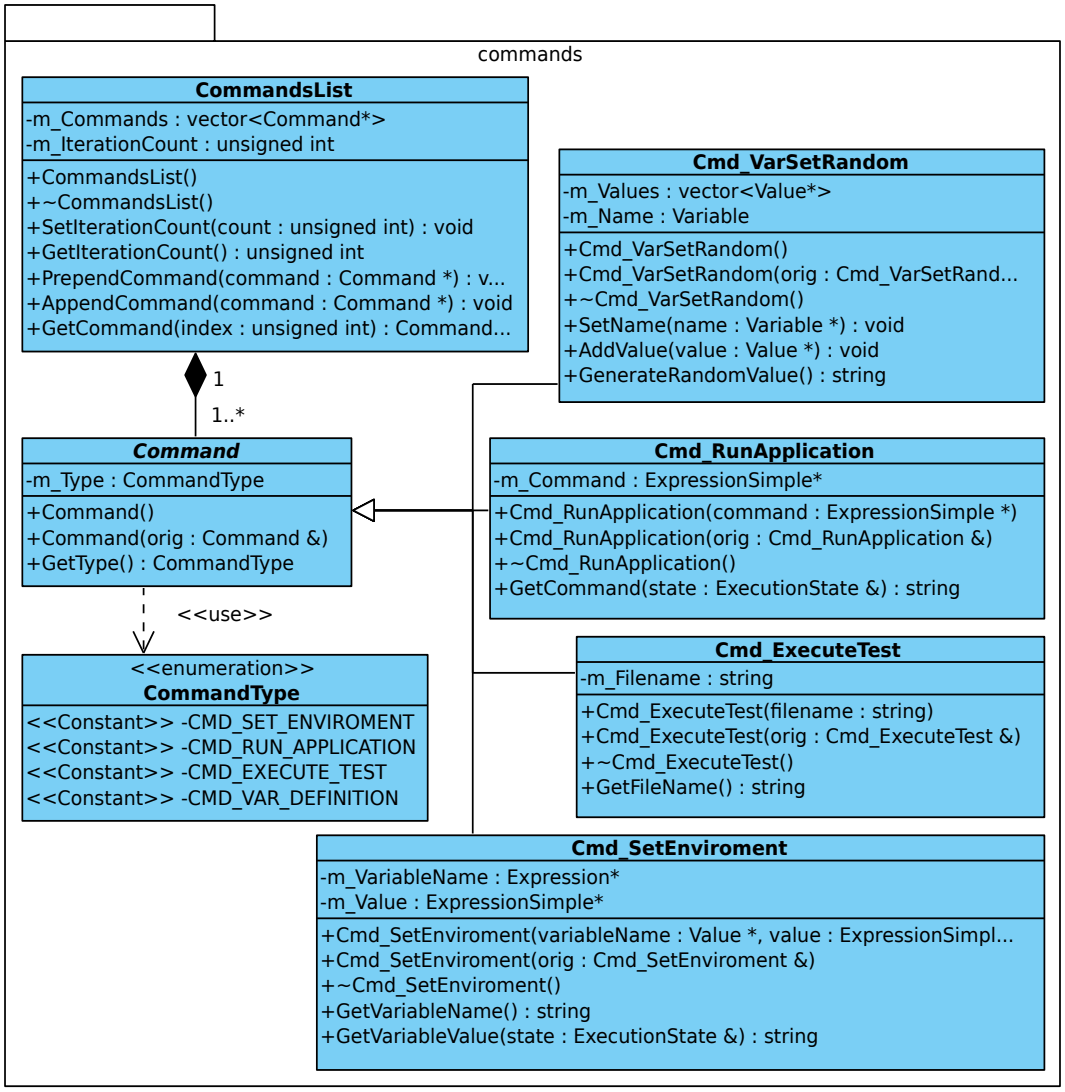
Modul generator

Generovanie výsledného zoznamu riadiacich príkazov a sekvencií udalostí je riadené modulom generátor, uvedeného na obrázku 4.8 . Jeho jadro tvorí trieda `Generator`, ktorá na základe internej reprezentácie popisu testovacej sady vygeneruje výslednú testovaciu sadu.

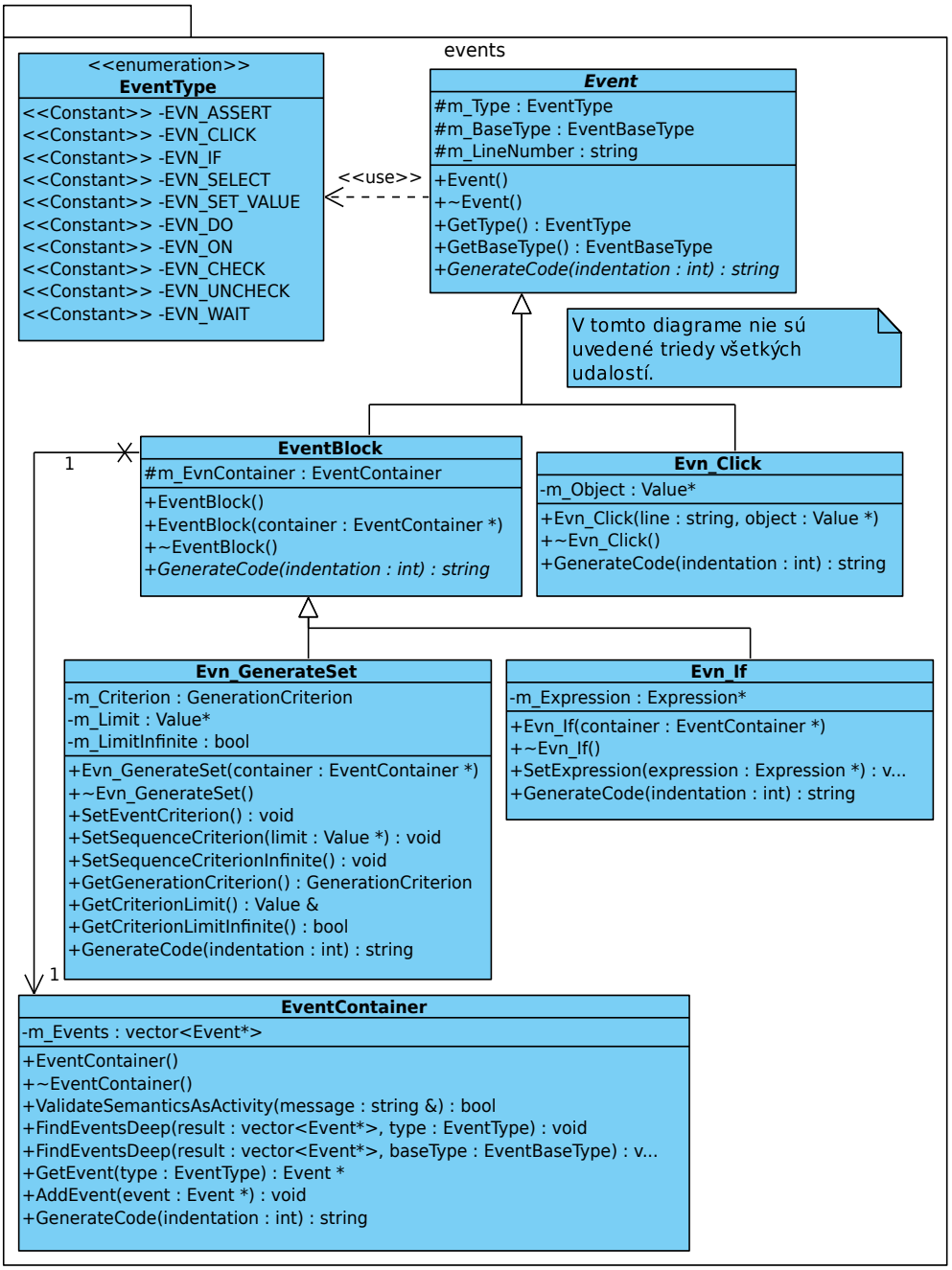
Modul `cmdlib.py` obsahuje knižnicu príkazov jazyka pre popis testovacej sady. V nej sa



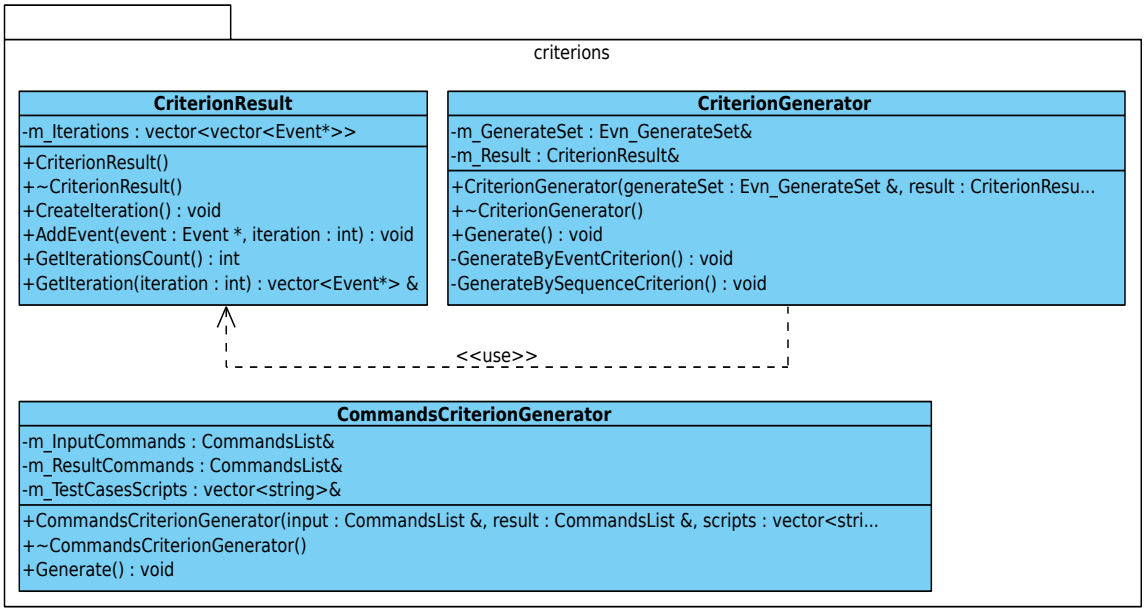
Obr. 4.4: Diagram tried modulu parsed-script



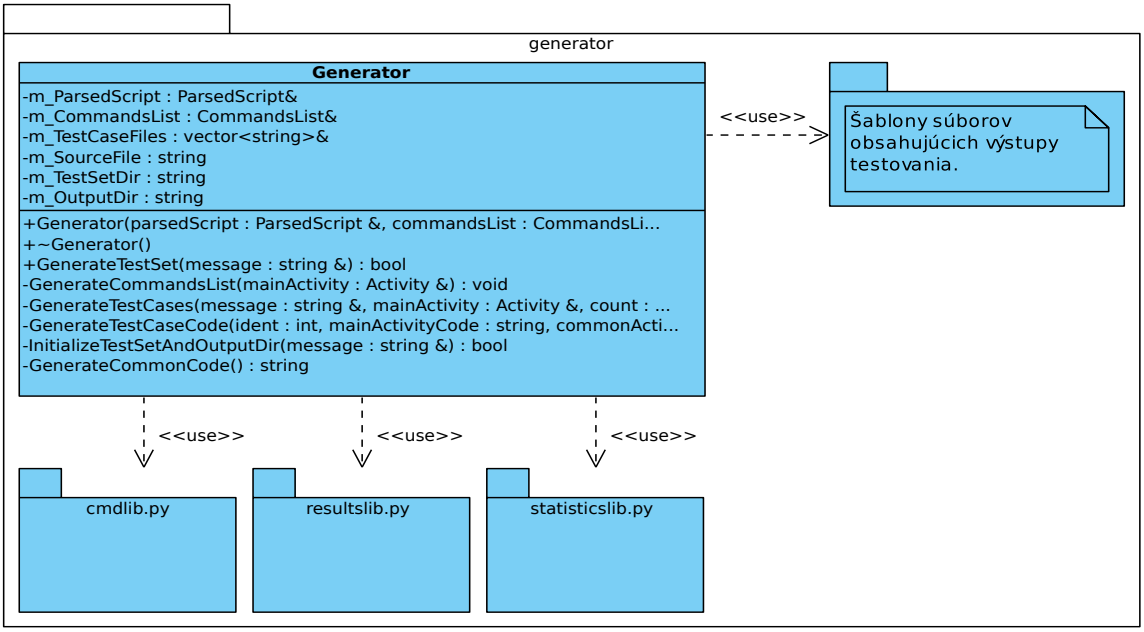
Obr. 4.5: Diagram tried modulu commands, ktorý je súčasťou modulu parsed-script



Obr. 4.6: Diagram tried modulu events, ktorý je súčasťou modulu parsed-script



Obr. 4.7: Diagram tried modulu criterions



Obr. 4.8: Diagram tried modulu generator

nachádza interpretácia týchto príkazov na zodpovedajúce príkazy systému LDTP. V moduloch `resultslib.py` a `statisticslib.py` sú implementované funkcie na zber údajov o priebehu a pokrytí vykonaných testov.

Okrem uvedených modulov sa tu nachádzajú šablóny súborov, ktoré budú obsahovať výstupy testovania. Jedná sa o súbory `text_result.xml` a `ldtp.log`. Modul generator tiež zabezpečuje prípravu priečinku do ktorého sa vygeneruje testovacia sada a priečinku v ktorom budú výstupy testovania.

Testovacia sada

Vygenerovaná testovacia sada pozostáva z testovacích prípadov, pričom každý testovací prípad je reprezentovaný jedným skriptom s názvom `test-case-*.py`. Znak `*` je nahradený príslušným číslom testovacieho prípadu. Testovacia sada obsahuje vždy aspoň jeden testovací prípad.

V rámci testovania je možné testovacie prípady spúšťať opakovane, s rôznymi parametrami testovanej aplikácie alebo s rôznymi hodnotami premenných prostredia, v ktorom bude aplikácia testovaná. Testovacie skripty jednotlivých testovacích prípadov sú tvorené kódom aktivít z popisu testovacej sady a kódom, implementujúcim ošetrovanie vzniknutých chýb a zber údajov o vykonanom testovaní.

Modul test-engine

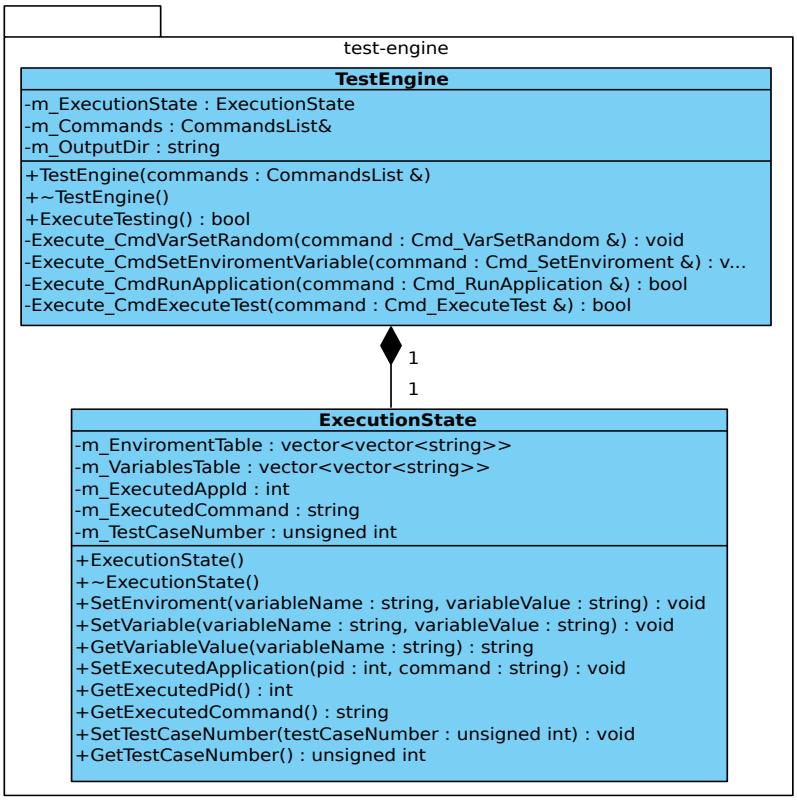
Vykonávanie vygenerovanej testovacej sady je riadené modulom test-engine, ktorého diagram tried je na obrázku 4.9 . Trieda `TestEngine` implementuje vykonávanie a funkcionalitu všetkých riadiacich príkazov. Vykonávaním týchto príkazov tak nastavuje prostredie testovanej aplikácie a spúšťa nad ňou testovacie prípady vygenerovanej testovacej sady. Pri svojej činnosti využíva triedu `ExecutionState`, ktorá uchováva aktuálne parametre prostredia testovanej aplikácie, identifikáciu práve testovanej aplikácie, vykonávaný testovací prípad a ďalšie údaje. Tento modul ďalej spracováva zozbierané informácie o priebehu a pokrytí jednotlivých testov, a generuje tak výstup systému.

4.2.3 Výstupy systému

Výstupom systému pre automatizované testovanie GUI sú tri súbory obsahujúce údaje o priebehu, výsledkoch a pokrytí testovania. Prvým je súbor `text_result.xml`. Ten obsahuje informácie o každom vykonaní testovacieho prípadu, ku ktorému sa zaznamenávajú: parametre testovanej aplikácie, nastavenie prostredia, priebeh a výsledky testovania. Každému vykonaniu testovacieho prípadu je priradený identifikátor, ktorý s ním umožňuje spárovať údaje o pokrytí testovania, uvedené v súbore `test_coverage.xml`.

Popis nastavenia prostredia obsahuje názvy a hodnoty premenných prostredia, ktoré boli zadané užívateľom. Priebeh a vykonanie testovacieho prípadu je popísané prostredníctvom záznamov o začatí a ukončení vykonávania jednotlivých aktivít a samotného testovacieho prípadu. Výsledky testovania sú tvorené záznamami vykonania príkazov `Evn_Assert` a vzniku prípadných výnimiek. Všetky tieto informácie sú zaznamenané v rovnakom poradí, v akom počas testovacieho behu vznikli a obsahujú časovú značku udávajúcu moment ich zaznamenania.

V rámci vykonania príkazu `Evn_Assert` sa ukladá výsledná hodnota jeho výrazu, jeho textová správa a jeho zdroj. Zdroj tohto príkazu obsahuje názov príslušného súboru s popi-



Obr. 4.9: Diagram tried modulu test-engine

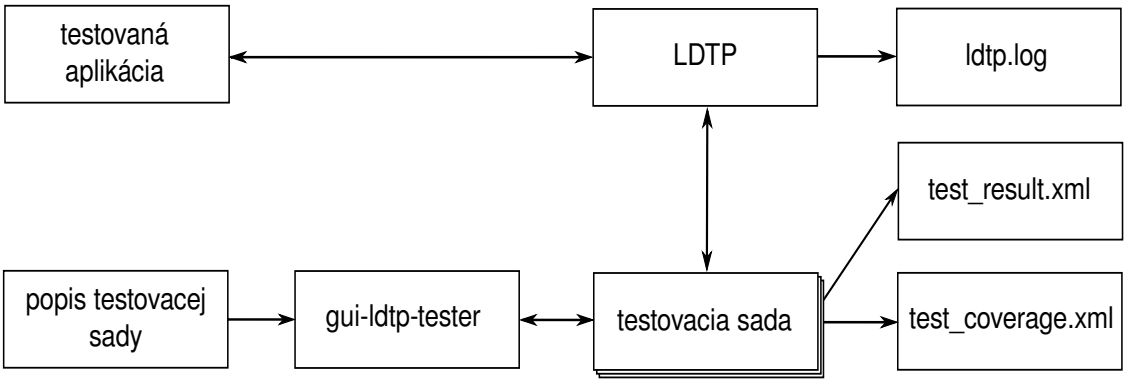
som testovacej sady a číslo riadku, na ktorom sa tento príkaz nachádza. Pri vzniku výnimky je zaznamenaný jej popis, *traceback* vygenerovaný interpretrom jazyka Python a jej zdroj.

Ďalším výstupom je súbor `test_coverage.xml`, ktorý obsahuje informácie o pokrytí v rámci jednotlivých vykonaní testovacích prípadov a v rámci celej testovacej sady. Uchováva sa tu zoznam aktivít z popisu testovacej sady a údaje o počte vykonaní jednotlivých aktivít v rámci testovania. Okrem toho je tu uložený zoznam komponent GUI, ktoré sú počas testovania viditeľné systémom LDTP. Tieto komponenty sú zoskupené do okien, v ktorých sa počas testovania nachádzajú. Ku každej komponente sa ukladá údaj o tom, koľkokrát bola v rámci testovania použitá. Napokon sa ešte počítajú údaje o percentuálnom pokrytí testovania a počte aktivít a komponent, ktoré otestované neboli. Štruktúra týchto súborov je podrobne popísaná v časti 5.2.1 .

Posledným výstupom tohto systému je súbor `ldtp.log`. Tento súbor obsahuje záznam správ systému LDTP, ktoré tento systém vygeneroval na základe jeho používania počas testovania.

4.3 Činnosť systému

V tejto časti je uvedený popis činnosti systému pre automatizované testovanie GUI. Najskôr je popísaná komunikácia jeho častí, po ktorej nasleduje podrobný popis činnosti jednotlivých modulov systému, špecifikovaných v časti 4.2 . Na záver je uvedený sekvenčný diagram jazyka UML, ktorý znázorňuje spoluprácu modulov tohto systému.



Obr. 4.10: Komunikácia častí systému

4.3.1 Komunikácia v rámci systému

Popis komunikácie v rámci systému je uvedený na obrázku 4.10. Činnosť systému začína spracovaním súboru obsahujúceho popis testovacej sady programom `gui-ldtp-tester`. Tento program vygeneruje testovaciu sadu, ktorá pozostáva z testovacích skriptov, spustí testovanú aplikáciu a tieto skripty začne postupne vykonávať. Pri ich vykonávaní sa využíva systém LDTP pre manipuláciu s GUI testovanej aplikácie, ale tiež aj pre získavanie informácií o stave testovaného GUI. Systém LDTP zapisuje svoje správy do súboru `ldtp.log` pre ich prípadnú analýzu užívateľom. Testovacie skripty okrem toho zbierajú informácie o priebehu, výsledkoch a pokrytí vykonaných testov. Tieto informácie sa nakoniec sumarizujú v súboroch `test_result.xml` a `test_coverage.xml`.

4.3.2 Popis činnosti modulov

Činnosť celého systému začína spustením programu `gui-ldtp-tester`. Ten musí mať pri spustení zadaný povinný parameter, ktorým je súbor obsahujúci popis testovacej sady. Nepovinné parametre sú: priečinok pre vygenerovanie testovacej sady a priečinok pre uloženie výsledkov testovania. Program počas svojej činnosti nemaže obsah týchto priečinkov, a preto je vhodné aby boli zadané priečinky prázdne. Po spustení programu `gui-ldtp-tester` sa najskôr štandardným spôsobom spracujú vstupné parametre programu. Spracovanie parametrov spočíva v kontrole zadania povinného parametru a uloženia ich hodnôt.

Modul parser

Po spracovaní vstupných parametrov je zahájená činnosť modulu parser. Ten si načíta súbor zadaný vstupným parametrom a vykoná jeho lexikálnu a syntaktickú analýzu. Počas nej tiež vytvára internú reprezentáciu spracovávaného popisu testovacej sady. Ak modul objaví syntaktickú chybu, zobrazí užívateľovi chybovú správu a činnosť systému končí. Správa o nájdenej chybe obsahuje číslo riadku spracovávaného súboru, pri ktorom chyba vznikla. Je to z dôvodu jej ľahšej identifikácie. Výstupom tohto modulu je usporiadaný strom objektov tried modulu parsed-script, ktorý plní úlohu internej reprezentácie spracovaného popisu.

Modul parsed-script

Tento modul najskôr vykoná sémantickú kontrolu popisu testovacej sady. Sémantika tohto popisu je kontrolovaná na usporiadanom strome najskôr v rámci celého skriptu, potom

v rámci jednotlivých aktivít a nakoniec v rámci blokových udalostí. Pri kontrole sa overuje, či popis spĺňa podmienky uvedené v časti 4.1.5. V prípade že je objavená sémantická chyba, program zobrazí užívateľovi chybovú správu a činnosť systému je ukončená.

Modul generator a modul criterions

Riadenie je po skončení sémantickej validácie predané modulu generator. Tento modul zaháji svoju činnosť prípravou priečinku pre vygenerovanie testovacej sady. Do neho skopíruje súbor `cmdlib.py` obsahujúci implementáciu príkazov jazyka pre popis testovacej sady, súbor `resultslib.py` implementujúci zber údajov o priebehu a výsledkoch testovania, a súbor `statisticslib.py` implementujúci zber údajov o pokrytí vykonaného testovania. Tieto súbory sú využívané vygenerovanými skriptami, tvoriacimi testovaciu sadu, a preto musia byť v spoločnom priečinku.

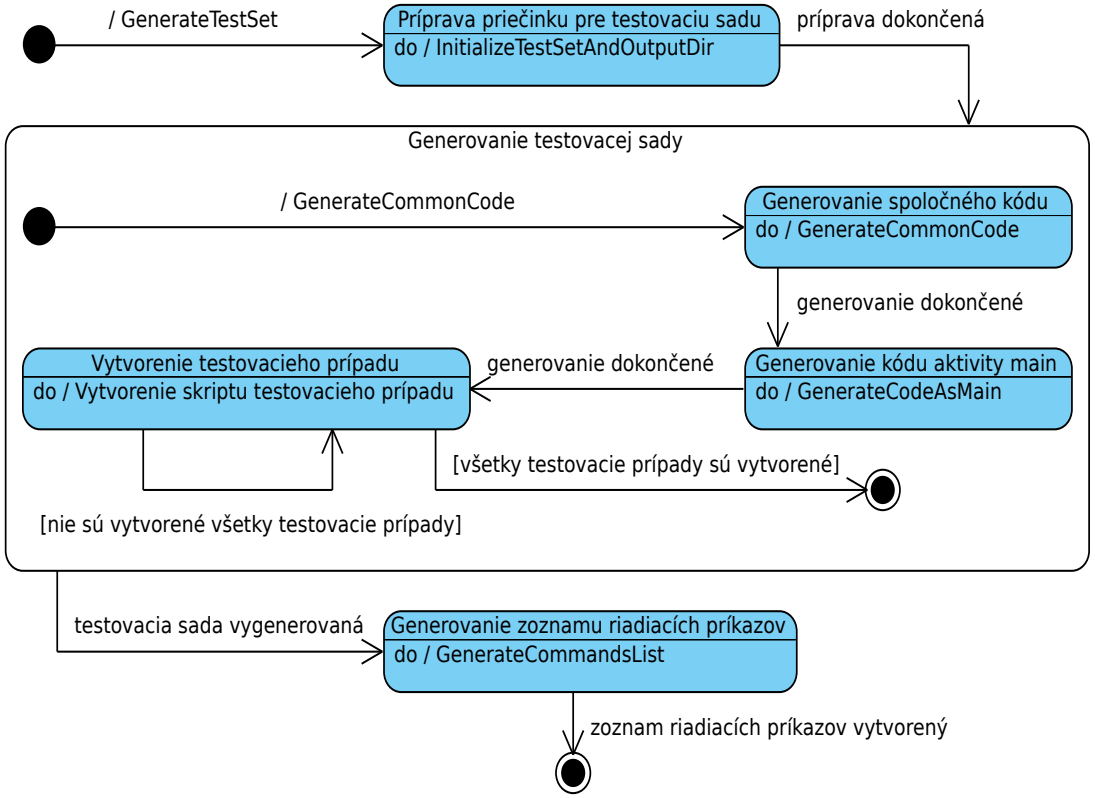
Do priečinka pre uloženie výstupov testovania skopíruje šablóny výstupných súborov. Jedná sa o súbor `test_result.xml`, kam budú ukladané informácie o priebehu a výsledkoch testov a súbor `ldtp.log`, do ktorého sa budú ukladať správy systému LDTP. Nakoniec vytvorí základnú štruktúru súboru `test_coverage.xml`, ktorá sa bude postupne aktualizovať a bude obsahovať informácie o pokrytí testov.

Po príprave priečinkov sa zaháji samotné generovanie testovacej sady. Najskôr sa vygenerujú jednotlivé testovacie skripty, ktoré implementujú jednotlivé testovacie prípady a spolu tvoria testovaciu sadu. Samotné generovanie ich obsahu je rozdelené na niekoľko častí, ktoré sú implementované v module parsed-script. Najskôr sa vygeneruje kód aktivít, ktoré nemajú identifikátor `main` a ich kód bude rovnaký pre všetky testovacie prípady. Potom sa vygeneruje kód aktivity `main` pre každý testovací prípad.

Kódy jednotlivých aktivít sú generované nasledovne. Príkaz pre vygenerovanie sekvencie udalostí sa rozgeneruje na výsledné sekvencie a tie sa spolu s kódom ostatných udalostí vložia do výsledného kódu aktivity. Každá udalosť je v tomto kóde reprezentovaná volaním príkazu z knižnice príkazov `cmdlib.py`. Takto vygenerované úseky kódu sa vložia do výsledného kódu, ktorý implementuje vykonanie aktivity `main`, prácu s výstupnými súbormi a ošetrovanie chýb. Vygenerované testovacie skripty uloží modul generator do priečinka definovaného vstupným parametrom.

Pri generovaní testovacích prípadov dochádza ku generovaniu sekvencií udalostí v jednotlivých aktivitách. Sekvencie udalostí sú generované modulom criterions. Tento modul obsahuje triedy, ktorých objekty dokážu nezávisle generovať sekvencie udalostí z jednotlivých príkazov `generate-set`. Každý objekt vygeneruje sekvenciu udalostí z príslušného príkazu na základe príslušného kritéria. Výsledná sekvencia je potom používaná ďalej pri generovaní kódu testovacích prípadov. Podrobný popis generovania sekvencií udalostí je uvedený v časti 4.4.

Modul generator nakoniec vygeneruje výsledný zoznam riadiacich príkazov, ktoré popisujú vykonávanie celej testovacej sady. Výsledný zoznam je tvorený niekoľkými kópiami zoznamu riadiacich príkazov zadaného v popise testovacej sady. Pre každý vygenerovaný testovací prípad je vytvorená jedna kópia zoznamu príkazov, ktorá počas testovania zabezpečí priradenie hodnoty definovaným premenným, nastavenie vlastného prostredia, spustenie aplikácie s požadovanými parametrami a vykonanie príslušného testovacieho prípadu. Na koniec každej kópie zoznamu príkazov je doplnený implicitný riadiaci príkaz `Cmd_ExecuteTest`, ktorý zabezpečí spustenie testovacieho prípadu. Stavový diagram popisujúci činnosť modulu generator je uvedený na obrázku 4.11.



Obr. 4.11: Stavový diagram činnosti modulu generator

Modul test-engine

Vygenerovaním testovacej sady končí činnosť modulu generator a začína činnosť modulu test-engine. Ten pri svojej činnosti ukladá informácie o aktuálnom stave vykonávania testovacej sady do objektu triedy `ExecutionState`. Tento stav si modul test-engine najskôr inicializuje, a potom začne postupne vykonávať riadiace príkazy vo vygenerovanom zozname.

Ak popis testovacej sady obsahuje generovanie zoznamu riadiacich príkazov podľa kritéria `random-data-criterion`, bude výsledný zoznam riadiacich príkazov vykonaný viac krát, podľa zadaného limitu. To znamená, že všetky testovacie prípady budú vykonané viac krát, ale vždy s rôznymi parametrami prostredia a aplikácie. Ak je limit zadaný znakom `*`, výsledný zoznam príkazov sa bude vykonávať nekonečnom cykle. Jeho vykonávanie musí prerušiť užívateľ zaslaním signálu `SIGHUP`, `SIGINT` alebo `SIGTERM` procesu, ktorý je inštanciou spusteného programu `gui-ldtp-tester`. Po obdržaní signálu sa dokončí vykonávanie aktuálneho testovacieho prípadu a testovanie sa ukončí.

Modul test-engine implementuje vykonávanie všetkých riadiacich príkazov. Ich činnosť vyzerá nasledovne. Príkaz `Cmd_VarSetRandom` nastaví pri svojom vykonaní hodnotu príslušnej premennej na náhodne vybranú hodnotu z príslušného zoznamu. Keďže hodnota sa vyberá až v rámci tohto príkazu, premennej je možné priradiť náhodnú hodnotu pred vykonaním jednotlivých testovacích prípadov. Výsledná hodnota sa uloží do aktuálneho stavu vykonávania testovacej sady.

Príkaz `Cmd_SetEnviromentVariable` si najskôr zistí hodnotu, ktorú má nastaviť do príslušnej premennej prostredia. Hodnota môže byť v rámci tohto príkazu zadaná buď

priamo alebo názvom definovanej premennej. V prípade že je zadaná premennou, zistí sa jej náhodne vygenerovaná hodnota z aktuálneho stavu vykonávania aplikácie. Nakoniec sa naspäť do tohto stavu uloží názov príslušnej premennej prostredia a jej nová hodnota.

Samotné spustenie testovanej aplikácie je zabezpečené príkazom `Cmd_RunApplication`. Tento príkaz najskôr získa reťazec špecifikujúci testovanú aplikáciu a jej parametre. Reťazec môže byť opäť zadaný priamo alebo pomocou definovanej premennej. Ak je zadaný prostredníctvom premennej, zistí sa jej hodnota z aktuálneho stavu vykonávania testovacej sady. Tento reťazec je ďalej spracovaný, čím sa zistí cesta k spustiteľnému súboru testovanej aplikácie a jej parametre. Ak tento reťazec neobsahuje cestu k spustiteľnému súboru aplikácie, užívateľovi sa zobrazí chybová správa a činnosť systému končí.

Ďalej začína proces spúšťania testovanej aplikácie. Modul test-engine najskôr vytvorí nový proces, ktorému nastaví premenné prostredia podľa predchádzajúcich riadiacich príkazov. Názvy a hodnoty premenných prostredia sú získané z aktuálneho stavu vykonávania testovacej sady. Hodnoty premenných prostredia sú nastavované v poradí, v ktorom boli uvedené aj príslušné riadiace príkazy v rámci zoznamu. Po nastavení prostredia je obraz tohto procesu nahradený obrazom procesu testovanej aplikácie so zadanými parametrami. Týmto spôsobom je aplikácia spustená a po uplynutí timeoutu pre spustenie aplikácie je považovaná za pripravenú na testovanie. Jej identifikácia je uložená do aktuálneho stavu vykonávania testovacej sady.

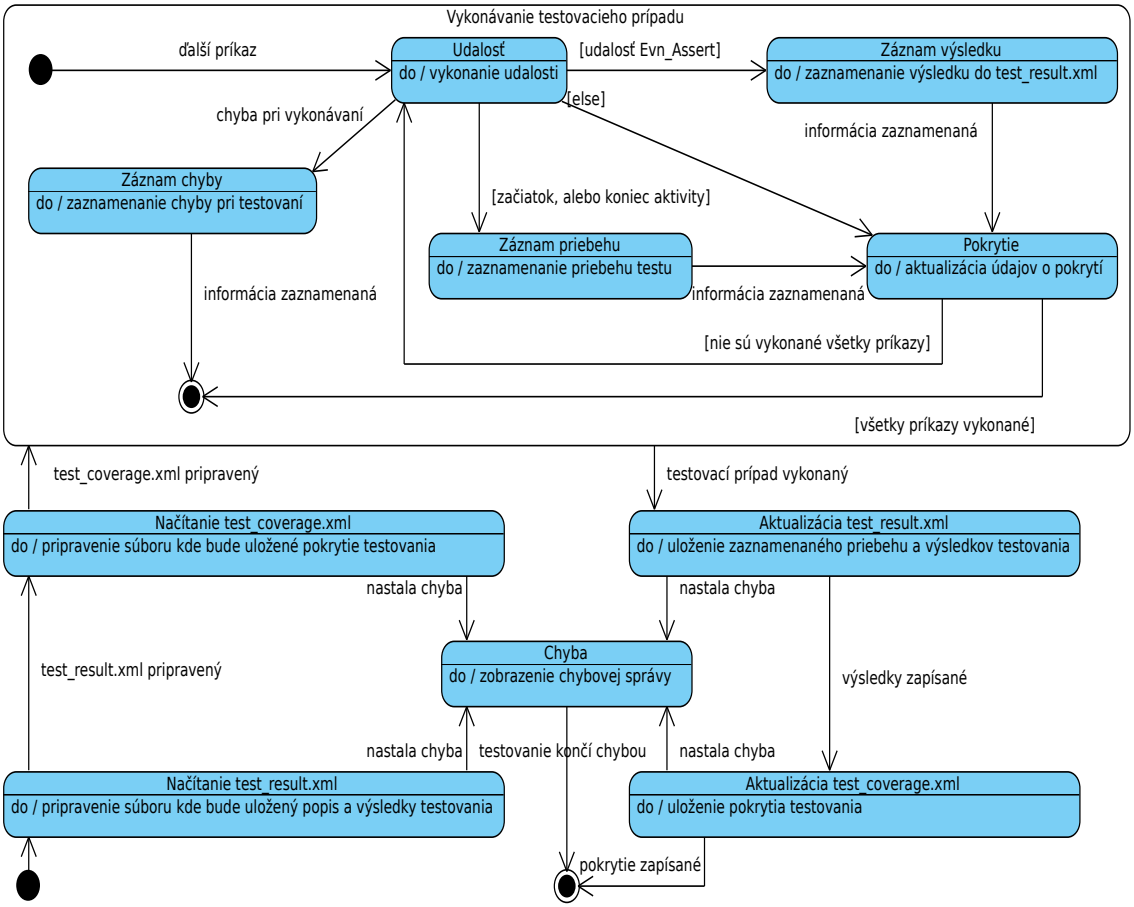
Ak počas nastavovania prostredia alebo spúšťania aplikácie nastane chyba, užívateľovi sa zobrazí chybová správa a vytvorený proces je ukončený. Hlavný proces síce pokračuje v testovaní tým, že spustí vykonávanie príslušného testovacieho prípadu, no tento hneď na začiatku skončí chybou, keďže sa mu neporadí nájsť okno testovanej aplikácie. Testovanie tak pokračuje spustením ďalšieho testovacieho prípadu.

Posledným príkazom v rámci testovacieho behu je riadiaci príkaz `Cmd_ExecuteTest`. Vykonanie tohto príkazu začína prípravou súboru `test_result.xml`. Táto príprava zahŕňa vytvorenie XML štruktúry, do ktorej sa neskôr vloží priebeh a výsledky vykonania testovacieho prípadu. Rovnakým spôsobom sa pripraví aj súbor `test_coverage.xml`, do ktorého budú neskôr vložené informácie o pokrytí vykonaného testovacieho prípadu a súbor `ldtp.log`.

Nasleduje samotné spustenie skriptu obsahujúceho príslušný testovací prípad. Tento skript je spustený tak, aby sa výstup systému LDTP automaticky ukladal do súboru `ldtp.log`. Týmto sa predá riadenie skriptu príslušného testovacieho prípadu, čím sa vykoná testovanie. Po jeho ukončení sa riadenie vráti naspäť modulu test-engine, ktorý aktualizuje obsah súboru `test_coverage.xml`. Vtedy sa informáciami o pokrytí vykonaného testovania aktualizujú informácie o pokrytí v rámci celej testovacej sady. Po tejto aktualizácií sa testovaná aplikácia ukončí. Ak počas vykonávania príkazu nastane chyba alebo vykonávanie testovacieho skriptu skončí s chybou, užívateľovi sa zobrazí chybová správa a následne sa ukončí beh testovanej aplikácie a činnosť celého systému.

Proces ukončovania testovanej aplikácie je nasledovný. Najskôr sa z aktuálneho stavu vykonávania testovacej sady zistí, ktorá aplikácia má byť ukončená. Potom sa z tohto stavu vymažú informácie o spustenej aplikácií, premenné prostredia a premenné definované v riadiacich príkazoch. Ešte pred samotným ukončením sa aplikácií poskytne čas na to, aby sa ukončila sama. Ak aplikácia stále beží, modul test-engine jej pošle signál `SIGTERM` a počas ďalšieho timeoutu jej umožní ukončiť sa. Ak aplikácia ani potom nie je ukončená, modul test-engine jej zašle signál `SIGKILL`, ktorý ju násilne ukončí.

Ak je aplikácia ukončená pred poslaním signálu `SIGKILL`, testovanie pokračuje ďalej. V opačnom prípade sa užívateľovi zobrazí chybová správa a ukončí sa beh systému. Činnosť



Obr. 4.12: Stavový diagram činnosti testovacej sady

modulu test-engine je definitívne ukončená po vykonaní posledného príkazu poslednej iterácie zoznamu riadiacich príkazov alebo po obdržaní jedného z vyššie uvedených signálov a ukončení aktuálneho vykonávania testovacieho prípadu. Vtedy končí aj činnosť celého systému.

Popis činnosti testovacej sady

Testovacia sada je tvorená testovacími prípadmi. Každý testovací prípad je implementovaný vo vlastnom vygenerovanom skripte. Činnosť jednotlivých skriptov je nasledovná. Skript dostane riadenie od modulu test-engine. Najskôr načíta súbor `test_result.xml` a pripraví sa na zaznamenávanie. Potom zo súboru `test_coverage.xml` načíta zoznam aktivít v popise testovacej sady. Následne začne vykonávanie aktivít príslušného testovacieho prípadu. Počas neho sa vykonávajú udalosti jednotlivých aktivít v poradí, v akom sú uvedené aj pri popise testovacej sady alebo v akom boli vygenerované. Na obrázku 4.12 je uvedený stavový diagram vykonávania vygenerovaného skriptu, ktorý reprezentuje jeden testovací prípad.

Počas testovania sa ukladajú informácie o začatí a ukončení vykonávania jednotlivých aktivít, o vykonaných príkazoch `Evn_Assert` a o vzniknutých výnimkách. V rámci každého príkazu `Evn_Assert` sa k údajom o priebehu a výsledkoch testovania uloží výsledná hodnota príslušného vyhodnoteného výrazu a príslušná správa. Ak počas vykonávania príkazu nastane výnimka, je zaznamenaná do údajov o priebehu testovania a vykonávanie aktuálneho

testovacieho prípadu skončí. Vzniknutá výnimka detekuje buď chybu objavenú v testovanom GUI alebo chybu pri práci so systémom LDTP. Jej výskyt síce spôsobí ukončenie aktuálneho vykonávania testovacieho prípadu, no činnosť systému pokračuje ďalej.

V rámci vykonávania jednotlivých príkazov sa aktualizujú informácie o pokrytí testovania. Pri začatí vykonávania aktivity sa aktualizuje údaj o počte jej spustení. Pri zmene kontextu na nové okno sa pomocou systému LDTP zistí zoznam komponent jeho GUI a údaje o pokrytí sa aktualizujú. Vždy keď je komponenta použitá ako parameter príkazu, aktualizuje sa údaj o počte testov v rámci ktorých bola použitá. Na záver vykonávania testovacieho prípadu sa vypočíta percentuálne pokrytie otestovaných aktivít a komponent GUI, a tiež počet neotestovaných aktivít a nepoužitých komponent.

Po skončení vykonávania testovacieho prípadu sa získané údaje zapíšu do príslušných súborov a riadenie je predané modulu test-engine. Ak pri načítaní alebo zápise dát do súborov obsahujúcich popis, výsledky a pokrytie testovania vznikne chyba, vykonávanie aktuálneho testovacieho prípadu skončí s chybou, čo vedie k ukončeniu činnosti celého systému.

Činnosť modelov systému, popísaná v tejto časti, je znázornená v sekvenčnom diagrame na obrázku 4.13 . Z dôvodu lepšej prehľadnosti je v tomto diagrame znázornená iba hlavná funkcionalita jednotlivých modulov a ich vzájomná spolupráca.

4.4 Automatické generovanie testovacej sady

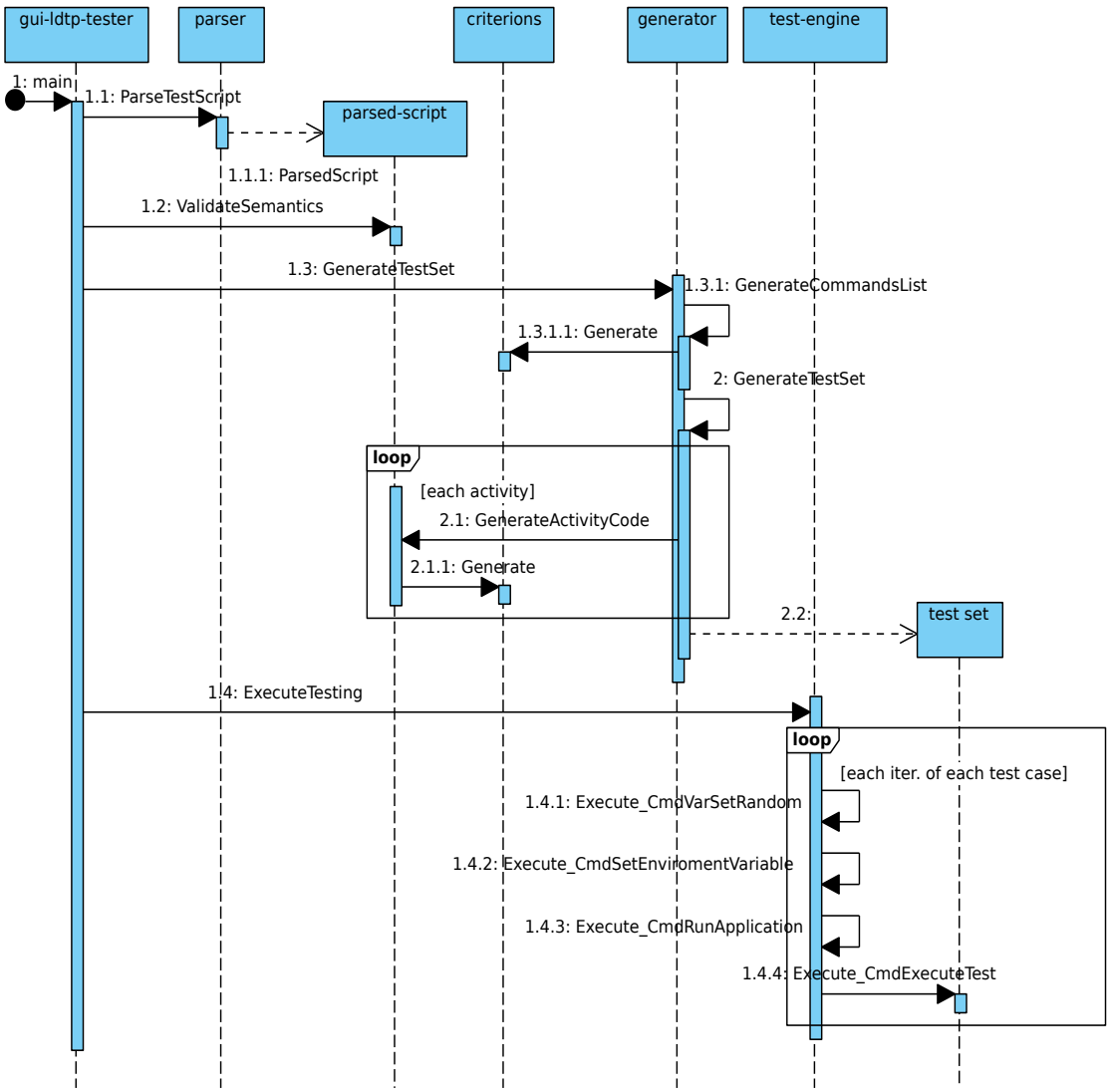
Systém pre automatizované testovanie GUI, popísaný v tejto kapitole, umožňuje automaticky generovať sekvencie udalostí a zoznamy riadiacich príkazov. Toto generovanie je špecifikované príkazom `Evn_GenerateSet`. Ten obsahuje zoznam udalostí alebo riadiacich príkazov, ktorých postupnosti majú byť generované a kritérium definujúce spôsob generovania. V tomto systéme som definoval tri kritériá generovania: `event-criterion`, `sequence-criterion` a `random-data-criterion`.

4.4.1 Generovanie zoznamu riadiacich príkazov

Zoznam riadiacich príkazov, uvedených na začiatku aktivity `main`, je možné generovať podľa kritéria `random-data-criterion`, ktorému sa zadáva limit určujúci počet opakovaní vykonávania tohto zoznamu.

Zoznam riadiacich príkazov je najskôr v rámci vytvárania testovacej sady niekoľkokrát skopírovaný do výsledného zoznamu. Každá táto kópia umožňuje vykonať jeden testovací prípad s konkrétnymi nastaveniami aplikácie, v konkrétnom prostredí. K použitiu uvedeného generovacieho kritéria prichádza až v module test-engine, ktorý urobí niekoľko iterácií vykonávania výsledného zoznamu riadiacich príkazov. Počet týchto iterácií je daný práve zadaným limitom. Ten môže mať formu celého čísla z uzatvoreného intervalu <1;100>, alebo môže byť zadaný znakom `*`. Znak `*` spôsobí, že výsledný zoznam sa bude opakovane vykonávať až pokým program `gui-ldtp-tester` neobdrží signál `SIGHUP`, `SIGINT` alebo `SIGTERM`.

Tento spôsob generovania je vhodný pri zadávaní parametrov riadiacich príkazov prostredníctvom definovaných premenných. Na začiatku zoznamu riadiacich príkazov je možné definovať premennú prostredníctvom zoznamu jej hodnôt. Pri každej iterácií je tejto premennej priradená náhodná hodnota z príslušného zoznamu. Týmto je možné vykonávať jednotlivé testovacie prípady s rôznymi parametrami testovanej aplikácie a v rôznom prostredí.



Obr. 4.13: Spolupráca modulov systému

4.4.2 Generovanie sekvencií udalostí

Sekvencie udalostí v aktivitách je možné generovať podľa kritérií `event-criterion` alebo `sequence-criterion`. Výsledkom generovania je usporiadaná množina usporiadaných sekvencií udalostí. Udalosti, z ktorých sa generujú výsledné sekvencie, sú zadané v tele príslušného príkazu `Evn_GenerateSet`. Poradie udalostí v rámci sekvencie a poradie sekvencií v rámci vygenerovanej množiny je dané poradím udalostí v tele príkazu `Evn_GenerateSet`. Toto je prehľadne znázornené v konkrétnych príkladoch uvedených ďalej.

Generovanie sekvencií udalostí je možné vykonať v rámci aktivity `main` alebo v rámci ostatných aktivít, pričom výsledná testovacia sada je v týchto prípadoch odlišná. Generovaním sekvencií udalostí v aktivite `main` sa určuje počet a základná štruktúra výsledných testovacích prípadov. Testovacia sada bude obsahovať toľko testovacích prípadov, koľko bude vygenerovaných sekvencií. Telo aktivity `main` bude v jednotlivých testovacích prípadoch odlišné. Príslušný príkaz `Evn_GenerateSet` sa totiž v každom testovacom prípade nahradí príslušnou sekvenciou udalostí. Zbytok aktivity zostáva v celej testovacej sade rovnaký.

Ak sa sekvencie udalostí generujú v ostatných aktivitách, telo danej aktivity bude vykonané viac krát za sebou. Počet opakovaní je daný počtom vygenerovaných sekvencií udalostí. V rámci každého opakovania však bude na mieste príkazu `Evn_GenerateSet` vykonaná príslušná sekvencia udalostí.

Kritérium `event-criterion` vygeneruje všetky sekvencie udalostí o dĺžke 1 a tým otestuje, že každá udalosť môže byť vykonaná aspoň raz. Použitie tohto kritéria je ukázané na nasledujúcej aktivite. Písmená A, B, C, D, E reprezentujú jednotlivé udalosti.

```
activity example
{
   A
   generate-set event-criterion
   {
      B
      C
      D
   }
   E
}
```

Pri použití kritéria bude aktivita `example` vykonaná trikrát. Jedno vykonanie aktivity je znázornené jednou usporiadanou množinou udalostí, vykonaných v rámci jej tela. Poradie vykonávania udalostí bude nasledovné:

```
{A, B, E}, {A, C, E}, {A, D, E}
```

Kritérium `sequence-coverage` vygeneruje všetky sekvencie zadanej dĺžky, čím otestuje vykonávanie daných udalostí v rôznych postupnostiach. V jednotlivých sekvenciách sa každá udalosť vyskytuje najviac jedenkrát a ich dĺžka je daná práve limitom kritéria. Limit je zadaný celým číslom, ktorého hodnota musí byť nezáporná a zároveň nesmie byť väčšia ako počet udalostí v príslušnom príkaze `Evn_GenerateSet`.

Ďalšou možnosťou je určiť limit kritéria znakom `*`. V tom prípade sa vygeneruje množina, ktorá bude obsahovať sekvencie udalostí postupne o dĺžke 0, 1, 2, až n, kde n je počet udalostí z ktorých sa sekvencie generujú. Ako príklad uvádzam použitie tohto kritéria na nasledujúcej aktivite.

```
activity example
{
   A
   generate-set sequence-criterion 2
   {
      B
      C
      D
   }
   E
}
```

V tomto prípade bude aktivita `example` vykonaná trikrát. Postupnosť vykonávania udalostí bude nasledovná:

```
{A, B, C, E}, {A, B, D, E}, {A, C, D, E}
```

Ďalším príkladom použitia tohto kritéria je generovanie sekvencií udalostí v nasledujúcej aktivite.

```
activity example
{
   A
   generate-set sequence-coverage *
   {
      B
      C
      D
   }
   E
}
```

Aktivita `example` bude vykonaná osemkrát, pričom postupnosť udalostí bude nasledovná:

```
{A, E}, {A, B, E}, {A, C, E}, {A, D, E}, {A, B, C, E}, {A, B, D, E},
{A, C, D, E}, {A, B, C, D, E}
```

4.5 Kritériá pokrytia

Na základe výstupných údajov, získaných počas vykonávania testovacej sady, som definoval dve kritériá pokrytia testovania. V súbore `text_coverage.xml` sa nachádza zonam aktivít definovaných v testovacej sade a informácia o tom, koľkokrát boli počas testovania vykonané. Na základe týchto údajov som definoval prvé kritérium pokrytia – *pokrytie aktivít*. Toto kritérium ukazuje koľko definovaných čiastkových testov bolo v rámci testovania vykonaných. Pokrytie je plné v prípade, že boli všetky aktivity z popisu testovacej sady vykonané aspoň raz. Pokrytie aktivít poskytuje informáciu o tom, aký veľký objem funkcionality GUI bol testovaním pokrytý.

Druhé kritérium pokrytia som definoval na základe údajov o použití jednotlivých komponent GUI pri testovaní. V súbore `test_coverage.xml` sú uložené údaje o tom, koľkokrát boli jednotlivé komponenty testovaného GUI použité v rámci testovania. Toto kritérium sa nazýva *pokrytie komponent* a ukazuje počet komponent, ktoré boli pri testovaní použité

aspoň raz. V tomto prípade je pokrytie plné, ak bola v rámci testovania použitá každá komponenta, každého zobrazeného okna aplikácie, aspoň raz. Toto kritérium poskytuje informáciu o tom, aká veľká časť GUI bola testovaním pokrytá. Dôležitým údajom však nie je iba skutočnosť, či bola daná komponenta použitá, ale aj koľkokrát bola použitá. Ak bola použitá viac krát, je pravdepodobnejšie že bola súčasťou komplexnejšieho testu.

Počas testovania sa zaznamenávajú všetky komponenty testovaného GUI, viditeľné systémom LDTP. Patria sem aj komponenty, ktorých identifikátor je vo výstupnom súbore tvorený prefixom `ukn`, čo znamená že ich typ je neznámy. Taktiež sa zaznamenávajú komponenty, ktoré nie je možné otestovať týmto systémom, keďže jazyk pre popis testovacej sady poskytuje iba podmnožinu funkcionality systému LDTP. Niektoré typy komponent ako napríklad komponenty typu *Filler* slúžiace na zarovnanie pozície ostatných komponent, môžu byť zo svojej podstaty otestované iba minimálne. Napríklad či existujú alebo sú viditeľné. Uvedené typy komponent znepresňujú informácie o pokrytí komponent, keďže plné pokrytie často nemôže byť v rámci testu dosiahnuté. Tieto netestovateľné komponenty sú vo výstupných údajoch uvedené preto, aby bol užívateľ informovaný o ich existencii a to aj za cenu znepresnenia výslednej hodnoty pokrytia podľa daného kritéria.

Presnosť tohto kritéria je negatívne ovplyvnená aj tým, že jedna komponenta môže mať počas testovania rôzne názvy a môže byť identifikovaná rôznymi textovými reťazcami. V takom prípade je nový názov komponenty počítaný ako nová komponenta. Pokrytie podľa popísaných kritérií sa systémom vyhodnocuje v rámci celej testovacej sady a tiež aj v rámci jednotlivých vykonaní testovacích prípadov.

Kapitola 5

Implementačné detaily

Táto kapitola obsahuje popis niektorých aspektov implementačnej úrovne, navrhnutého systému pre automatizované testovanie GUI.

5.1 Program gui-ldtp-tester

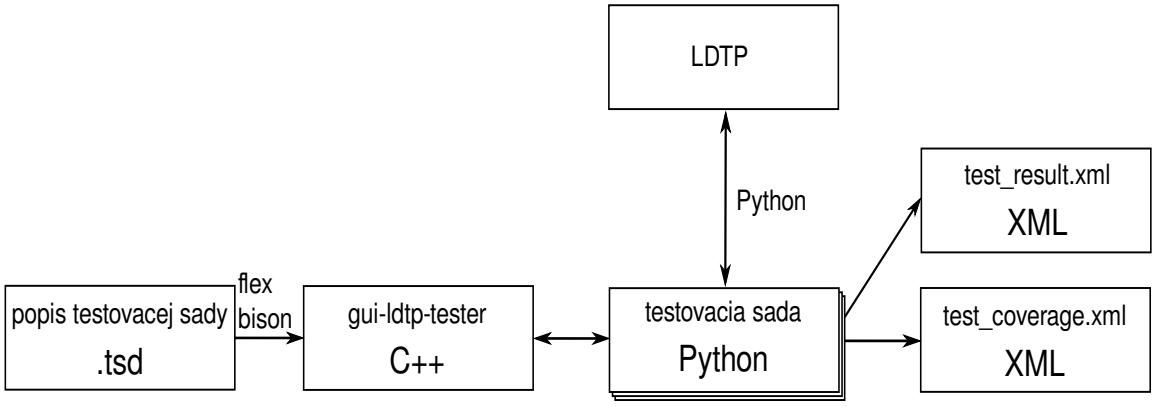
Základ celého systému pre automatizované testovanie GUI je tvorený konzolovou aplikáciou `gui-ldtp-tester`. Tento program riadi preklad popisu testovacej sady, generovanie testovacej sady, testovanie a generovanie výstupov testovania. Môže byť spustený s parametrami:

- `-t` – povinný parameter, ktorého argument určuje súbor s popisom testovacej sady,
- `-h` – parameter pre výpis nápovedy,
- `-o` – parameter, ktorého argument určuje priečinok pre uloženie výstupov testovania a
- `-s` – parameter, ktorého argument určuje priečinok pre uloženie vygenerovanej testovacej sady.

V prípade, že nie je zadaný priečinok pre uloženie testovacej sady alebo priečinok pre uloženie výstupov, použije sa aktuálny priečinok, v ktorom je program spustený. Uvedené priečinky by mali byť prázdne, pretože `gui-ldtp-tester` ich obsah pred použitím nevymaže.

Tento program bol vyvíjaný pod 64 bitovou verziou operačného systému Ubuntu 12.04, s jadrom systému Linux verzie 3.2.0 . Boli ním testované štandardné aplikácie grafického prostredia GNOME 3.2.1 . V rámci systému je použitých niekoľko programovacích jazykov, ktoré sú spolu s príslušnými časťami tohto systému uvedené na obrázku 5.1 .

Konzolový program `gui-ldtp-tester`, ktorý tvorí základ celého systému je implementovaný v jazyku C++. V niektorých prípadoch sú v tomto programe použité tiež funkcie z knižnice jazyka C, ako napríklad vytváranie procesu testovanej aplikácie, posielanie signálov alebo spustenie testovacieho prípadu. Jedným zo vstupov tohto programu je súbor obsahujúci popis testovacej sady, zapísaný v špeciálnom jazyku pre popis testovacej sady, ktorý bol vytvorený v rámci tejto práce. Tento súbor je preložený prostredníctvom lexikálneho a syntaktického analyzátoru, ktoré sú vygenerované počas prekladu celého programu. Syntaktický analyzátor je vygenerovaný nástrojom *Bison*, podľa popisu gramatických pravidiel. Tento analyzátor pri svojej činnosti využíva lexikálny analyzátor, vygenerovaný nástrojom *flex*. Program `gui-ldtp-tester` vygeneruje testovaciu sadu, ktorá je zložená zo skriptov



Obr. 5.1: Programovacie jazyky použité v rámci systému

v jazyku Python. Tieto skripty využívajú systém LDTP prostredníctvom jeho rozhrania pre jazyk Python. Testovacia sada generuje výstup systému, ktorý je zapísaný v jazyku XML.

5.2 Formát výstupných XML súborov

K výstupom činnosti uvedeného systému pre automatizované testovanie GUI patria súbory `test_result.xml` a `test_coverage.xml`. Ich štruktúra bude v tejto časti popísaná prostredníctvom DTD – Document type definition.

5.2.1 Formát údajov o priebehu a výsledku testovania

Súbor `test_result.xml` obsahuje údaje o priebehu a výsledkoch jednotlivých vykonávaní testovacích prípadov. Štruktúra tohto XML dokumentu je nasledovná:

```
<!DOCTYPE test-set [
   <!ELEMENT test-set (test-run+)>
   <!ELEMENT test-run (enviroment,application,test-execution-trace)>
   <!ELEMENT enviroment (variable*)>
   <!ELEMENT variable EMPTY>
   <!ELEMENT application EMPTY>
   <!ELEMENT test-execution-trace (start,
                                   (activity-start|
                                    activity-end|
                                    assertion|
                                    exception)*,
                                  end)>
   <!ELEMENT start EMPTY>
   <!ELEMENT activity-start EMPTY>
   <!ELEMENT activity-end EMPTY>
   <!ELEMENT assertion (result&source&message)>
   <!ELEMENT result (#PCDATA)>
   <!ELEMENT source EMPTY>
   <!ELEMENT message (#PCDATA)>
   <!ELEMENT exception (source&message&ldtp-traceback)>
   <!ELEMENT ldtp-traceback (#PCDATA)>
   <!ELEMENT end EMPTY>
```

```
   <!ATTLIST test-run id CDATA #required>
   <!ATTLIST test-run test-case CDATA #required>
   <!ATTLIST variable name CDATA #required>
   <!ATTLIST variable value CDATA #required>
   <!ATTLIST application command CDATA #required>
   <!ATTLIST start timestamp CDATA #required>
   <!ATTLIST activity-start id CDATA #required>
   <!ATTLIST activity-start timestamp CDATA #required>
   <!ATTLIST assertion timestamp CDATA #required>
   <!ATTLIST source file CDATA #required>
   <!ATTLIST source line CDATA #required>
   <!ATTLIST exception timestamp CDATA #required>
   <!ATTLIST activity-end id CDATA #required>
   <!ATTLIST activity-end timestamp CDATA #required>
   <!ATTLIST end timestamp CDATA #required>
]>
```

XML uzol `test-set` popisuje vykonávanie celej vygenerovanej testovacej sady a je zložený z popisu jednotlivých vykonaní testovacích prípadov, označených uzlom `test-run`. V rámci každého vykonávania testovacieho prípadu sa ukladajú údaje o nastavení prostredia, parametroch testovanej aplikácie a vykonaní testovacieho prípadu. Nastavenie prostredia je uložené v uzle `enviroment`, testovaná aplikácia a jej parametre sú v uzle `application`, použitý testovací prípad je v atribúte `test-case` a popis vykonávania testovacieho prípadu je uvedený v uzle `test-execution-trace`. Okrem toho je ku každému vykonaniu testovacieho prípadu priradený identifikátor v atribúte `id`, ktorý umožňuje spárovať údaje o pokrytí zo súboru `test_coverage.xml`.

Priebeh vykonania testovacieho prípadu je tvorený údajmi o začatí a ukončení jeho vykonávania, o začatí a ukončení vykonávania jednotlivých aktivít, o vyhodnotení definovanej podmienky v rámci príkazu `Evn_Assert` a o výskyte výnimky. Informácie o začiatku a konci testovacieho prípadu sú v uzloch `start` a `end`. Informácie o začiatku a konci vykonávania aktivít sú v uzloch `activity-start` a `activity-end`. Vyhodnotenie definovanej podmienky obsahuje uzol `assertion` a informácie o prípadnej výnimke sú v uzle `exception`. Každý uvedený údaj obsahuje tiež čas jeho zaznamenania, uvedený v atribúte `timestamp` príslušného uzlu.

Záznam o vyhodnotení definovanej podmienky v uzle `assertion` obsahuje výslednú hodnotu príslušného výrazu v uzle `result`, príslušnú správu uloženú v uzle `message` a zdroj príkazu v uzle `source`. Údaje o zaznamenanej výnimke v uzle `exception` obsahujú jej popis uvedený v uzle `message`, traceback vygenerovaný interpretom jazyka Python v uzle `ldtp-traceback` a zdroj výnimky v uzle `source`.

V rámci uzlu `source` sa uchováva názov použitého súboru s popisom testovacej sady, uloženého v atribúte `file` a príslušné číslo riadku v atribúte `line`. Vďaka tomu je možné rýchlo identifikovať príkaz, ktorého vykonanie spôsobilo vznik výnimky alebo podmienku, ktorá nebola splnená.

5.2.2 Formát údajov o pokrytí testovania

Údaje o pokrytí testovania sú zaznamenané v súbore `test_coverage.xml`. Štruktúra tohto XML dokumentu je nasledovná:

```
<!DOCTYPE coverage [
   <!ELEMENT coverage (test-set&(test-run+))>
   <!ELEMENT test-set (activity-coverage,widget-coverage)>
   <!ELEMENT test-run (activity-coverage,widget-coverage)>
   <!ELEMENT activity-coverage (summary&(activity+))>
   <!ELEMENT widget-coverage (summary&(window*))>
   <!ELEMENT summary EMPTY>
   <!ELEMENT activity EMPTY>
   <!ELEMENT window (widget+)>
   <!ELEMENT widget EMPTY>

   <!ATTLIST test-run id CDATA #required>
   <!ATTLIST summary all CDATA #required>
   <!ATTLIST summary coverage CDATA #required>
   <!ATTLIST summary not-tested CDATA #required>
   <!ATTLIST activity id CDATA #required>
   <!ATTLIST activity count CDATA #required>
   <!ATTLIST window id CDATA #required>
   <!ATTLIST widget id CDATA #required>
   <!ATTLIST widget count CDATA #required>
]>
```

Pokrytie vykonaných testov je zaznamenané a vyhodnotené v rámci celej testovacej sady (uzol `test-set`) a v rámci jednotlivých vykonaní testovacieho prípadu (uzol `test-run`). Oba XML uzly majú rovnakú štruktúru. Informácie o pokrytí sa skladajú z pokrytia aktivít zaznamenanom v uzle `activity-coverage` a pokrytia komponent zaznamenanom v uzle `widget-coverage`.

Údaje o pokrytí aktivít obsahujú zoznam aktivít v popise testovacej sady, uložených v uzloch `activity` a vyhodnotenie pokrytia v uzle `summary`. V rámci každej aktivity je uložený jej identifikátor v atribúte `id` a počet jej vykonaní v atribúte `count`.

Pokrytie komponent GUI obsahuje zoznam komponent (uzol `widget`) viditeľných počas testovania a vyhodnotenie pokrytia v uzle `summary`. Jednotlivé komponenty sú zoskupené do príslušných okien testovanej aplikácie, reprezentovaných uzlami `window`. V rámci každej komponenty je zaznamenaný jej identifikátor v atribúte `id` a počet jej použití v rámci testovania, ktorý je v atribúte `count`.

Vyhodnotenie pokrytia, v uzle `summary`, má vždy rovnakú štruktúru. V atribúte `all` je uložený počet všetkých aktivít alebo komponent. Atribút `coverage` obsahuje hodnotu udávajúcu percentuálny podiel použitých aktivít alebo komponent. Atribút `not-tested` zaznamenáva počet nevykonaných aktivít alebo komponent, ktoré neboli v rámci testovania vôbec použité.

5.3 Generovanie sekvencií udalostí

Jazyk pre popis testovacej sady umožňuje zapísať generovanie sekvencií udalostí prostredníctvom príkazu `generate-set` a kritéria `sequence-criterion` so zadaným limitom. Generovanie týchto sekvencií je implementované nasledovnou rekurzívnou funkciou, ktorá vygeneruje všetky sekvencie o dĺžke zadanej parametrom `seqLen`, z udalostí zadaných parametrom `input`. V každej vygenerovanej sekvencií sa daná udalosť nachádza najviac jedenkrát. Kód tejto funkcie vyzerá nasledovne:

```
1  void Generate(Event* input[], unsigned seqLen, Event* buffer[])
2  {
3     if (input.size() < seqLen)
4     {
5        return;
6     }
7
8     if (seqLen == 0)
9     {
10       AddToResult(buffer);
11       return;
12    }
13
14    unsigned originInputSize = input.size();
15    for (unsigned i = 0; i < originInputSize; i++)
16    {
17       buffer.push_back(input.at(0));
18       input.pop_front();
19       Generate(input, seqLen - 1, buffer);
20       buffer.pop_back();
21    }
22
23    return;
24 }
```

Parametre uvedenej funkcie sú:

- `input` – aktuálne pole udalostí, ktoré ešte neboli pri generovaní aktuálnej sekvencie použité,
- `seqLen` – počet udalostí, ktoré je ešte potrebné v rámci aktuálnej sekvencie vygenerovať a
- `buffer` – pole udalostí, ktoré tvorí aktuálny stav generovanej sekvencie.

Výsledné sekvencie sa ukladajú do členskej premennej pomocou funkcie `AddToResult` (riadok 10). Táto funkcia skopíruje udalosti uložené v premennej `buffer` a pridá ich do výsledku ako novú sekvenciu.

Funkcia pre generovanie sekvencií sa používa tak, že do parametru `input` sa zadajú udalosti z príkazu `generate-set`, do parametru `seqLen` sa vloží hodnota limitu kritéria a parameter `buffer` je tvorený prázdnym poľom. Po začatí jej vykonávania sa najskôr skontroluje, či pole udalostí `input` obsahuje dostatočný počet prvkov pre vygenerovanie sekvencie zadanej dĺžky (riadok 3). Potom sa skontroluje, či už vygenerovaná sekvencia udalostí obsahuje požadovaný počet prvkov (riadok 8). Ak áno, obsah premennej `buffer` sa uloží ako nová sekvencia do výsledku (riadok 10). Ďalej sa pre všetky udalosti, ktoré ešte neboli použité pre generovanie, vykonajú nasledovné akcie. Prvá udalosť z poľa `input` sa vloží do premennej `buffer` ako ďalší prvok sekvencie, a tým sa vygeneruje jeden prvok aktuálnej sekvencie (riadok 17). Keďže táto udalosť už bola pri generovaní použitá, odstráni sa z poľa `input` (riadok 18). Následne sa rekurzívne vygeneruje sekvencia kratšia o jeden prvok, z poľa zostávajúcich udalostí (riadok 19). Použitý prvok sa nakoniec odstráni aj z premennej `buffer` (riadok 20).

Ak je limit kritéria zadaný znakom `*`, zavolá sa táto funkcia niekoľkokrát, pričom jej parameter `seqLen` bude mať postupne hodnoty: 0, 1, 2, . . . , n, kde n je počet udalostí v príkaze `generate-set`. Týmto sa zo zadaných udalostí vygenerujú všetky sekvencie všetkých požadovaných dĺžok.

5.4 Spúšťanie testovanej aplikácie

Nastavenie parametrov testovanej aplikácie a jej spustenie je špecifikované príkazom `run`, z jazyka pre popis testovacej sady. Aplikácia je spolu s jej parametrami zadaná prostredníctvom textového reťazca. Tento reťazec sa skladá z cesty k binárnemu súboru testovanej aplikácie a zoznamu jej prípadných parametrov pri spustení. Všetky údaje sú v reťazci oddelené medzerou. Príkaz `run` je implementovaný funkciou triedy modulu `test-engine`, ktorá vyzerá nasledovne:

```
1  bool TestEngine::Execute_CmdRunApplication(Cmd_RunApplication& cmd)
2  {
3     string param[] = Split(cmd.GetCommand(), ' ', param);
4     // check parsed parameters ...
5
6     int pid = fork();
7     if (pid < 0) // error ...
8     else if (pid == 0) // child process
9     {
10       if (!SetEnviromentVariables()) { exit(-1); }
11       execv(param[0].c_str(), param);
12       exit(-1);
13    }
14    else // parent process
15    {
16       m_ExecutionState.SetExecutedApplication(pid, cmd.GetCommand());
17       sleep(TIMEOUT_APP_START);
18    }
19
20    return true;
21 }
```

Uvedenej funkcii je ako parameter `cmd` zadaný príkaz `run`, ktorý má byť vykonaný. Po jej spustení sa najskôr sa spracuje textový reťazec (riadok 3), obsahujúci testovanú aplikáciu a jej parametre. Potom sa skontroluje či tento reťazec obsahoval aspoň cestu k binárnemu súboru aplikácie. Ak nie, testovanie končí chybou. Po kontrole sa vytvorí nový proces (riadok 6), ktorý bude reprezentovať testovanú aplikáciu. V ňom sa funkciou `SetEnviromentVariables` nastavia užívateľom špecifikované premenné prostredia na požadované hodnoty (riadok 10). Na samotné nastavenie hodnoty premenných prostredia je použitá funkcia `setenv`. Ak počas ich nastavovania vznikne chyba, vytvorený proces je ukončený. Nakoniec sa funkciou `execv` nahradí obraz aktuálneho procesu, obrazom procesu testovanej aplikácie s príslušnými parametrami (riadok 11). V prípade chyby je proces ukončený (riadok 12).

Po vytvorení procesu pre testovanú aplikáciu pokračuje pôvodný proces zaznamenaním informácií o práve spustenej aplikácií. Do objektu uchovávajúceho stav testovania sa zapíše číslo procesu spustenej aplikácie a jej parametre (riadok 16). Nakoniec proces počká na

uplynutie timeoutu pre spustenie testovanej aplikácie (riadok 17). Tento timeout má nastavenú hodnotu 3 sekundy, čo je približná doba spustenia testovaných aplikácií počas vývoja systému.

Ak pri vykonávaní popísanej funkcie nastala chyba, funkcia vráti hodnotu `false`. V opačnom prípade vráti hodnotu `true`. Ak nastane chyba v procese testovanej aplikácie, funkcia vráti hodnotu `true` a testovací prípad po svojom spustení zistí, že GUI danej aplikácie sa v systéme nenachádza.

V rámci príkazu `run` je možné zadávať parametre aplikácie podobne ako pri jej spustení v príkazovom riadku. Z dôvodu použitia funkcie `execv` nie je možné v rámci tohto príkazu používať nástroje príkazového riadku, ako napríklad rúra (znak `|`) alebo presmerovanie štandardného výstupu aplikácie.

5.5 Spúšťanie testovacieho prípadu

Testovanie GUI aplikácie je zložené z vykonávania jednotlivých vygenerovaných skriptov, reprezentujúcich testovacie prípady. Tieto skripty môžu byť vykonávané s rôznymi parametrami testovanej aplikácie a s rôznym nastavením prostredia.

Spustenie testovacieho prípadu je reprezentované príkazom `Cmd_ExecuteTest`, ktorý je počas generovania testovacej sady automaticky vložený za každý príkaz `run`. Jeho funkcionalita je implementovaná funkciou v module `test-engine`, ktorá vyzerá nasledovne:

```
1  bool TestEngine::Execute_CmdExecuteTest(Cmd_ExecuteTest& command)
2  {
3     // ...
4     // prepare XML files for output
5     // ...
6
7     string pyLog = m_OutputDir + UTL_PYTHON_LOG_FILE_NAME;
8     string pyCmd = "python " + command.GetFileName() + " 2>> " + pyLog;
9     int ret = system(pyCmd.c_str());
10    if (ret != 0) // error ...
11
12    // ...
13    // update XML output files
14    // ...
15
16    if (!TerminateTestedApplication())
17    {
18       return false;
19    }
20
21    return true;
22 }
```

Uvedená funkcia očakáva ako parameter príkaz typu `Cmd_ExecuteTest`. Tento príkaz obsahuje názov skriptu s testovacím prípadom, ktorý má byť vykonaný. Príkaz sa vykonáva až po spustení testovanej aplikácie, takže jeho implementácia predpokladá bežiacu testovaciu aplikáciu.

Pri vykonávaní príkazu sa najskôr pripravia potrebné štruktúry vo výstupných súboroch pre zaznamenanie priebehu vykonania testovacieho prípadu, zaznamenanie pokrytia a za-

znamenanie správ systému LDTP. Ak pri príprave týchto súborov nastane chyba, testovaná aplikácia sa ukončí a testovanie skončí chybou. V prípade úspešnej aktualizácie uvedených súborov sa pripraví text príkazu (riadky 7 a 8), ktorý spustí vykonávanie testovacieho prípadu prostredníctvom interpretera jazyka Python. Testovanie je zahájené funkciou `system`, ktorá vytvorený text vykoná ako príkaz (riadok 9).

Po skončení vykonávania testovacieho prípadu sa overí jeho návratová hodnota (riadok 10). Ak testovací prípad skončil s chybou, ďalšie testovanie sa zastaví. V opačnom prípade sa zase aktualizuje obsah výstupných súborov. Ak počas ich aktualizácie nastane chyba, testovanie skončí s chybou. Na záver sa ukončí testovaná aplikácia (riadok 16), keďže nemusela byť ukončená v rámci vykonávania testovacieho prípadu.

Ak počas vykonávania popísanej funkcie nastala chyba, funkcia vráti hodnotu `false` a testovanie sa ukončí. V opačnom prípade vráti funkcia hodnotu `true`. Vždy však musí ukončiť spustenú testovanú aplikáciu.

5.6 Ukončovanie testovanej aplikácie

Po každom vykonaní testovacieho prípadu je potrebné zaistiť ukončenie testovanej aplikácie. Táto môže byť ukončená v rámci popisu testovacej sady, na konci každého testovacieho prípadu. Ak však ukončená nie je alebo testovací prípad skončí predčasne, je potrebné túto aplikáciu ukončiť dodatočne. Proces ukončovania je implementovaný vo funkcii `TestEngine::TerminateTestedApplication()` modulu `text-engine`.

Ešte pred samotným ukončením aplikácie sa, z objektu uchovávajúceho stav testovania, odstránia informácie o ukončovanej aplikácií. Samotné ukončovanie potom prebieha v niekoľkých krokoch:

1. Modul test-engine najskôr umožní procesu testovanej aplikácie ukončiť sa. Do uplynutia timeoutu pre dobehnutie aplikácie sa funkciou `waitpid` testuje ukončenie procesu a zisťuje sa jeho stav.

2. Ak aplikácia stále beží, systém jej pošle signál `SIGTERM` pre ukončenie. Potom pomocou funkcie `waitpid` zisťuje či proces aplikácie stále beží, až pokým neuplynie timeout pre ukončenie aplikácie.

3. Ak aplikácia stále nie je ukončená, pošle sa jej procesu signál `SIGKILL`, ktorý ju ukončí násilne. Systém potom do uplynutia timeoutu pre násilné ukončenie aplikácie opäť testuje ukončenie jej procesu, aby sa zabránilo vzniku *zombie procesov*.

Ak je aplikácia úspešne ukončená pred poslaním signálu `SIGKILL`, funkcia ukončovania vráti hodnotu `true` a testovanie pokračuje. V opačnom prípade vráti funkcia hodnotu `false` a testovanie je ukončené s chybou. Uvedené timeouty majú v systéme nastavenú hodnotu 2 sekundy. Táto hodnota bola zvolená na základe pozorovania doby ukončovania testovaných aplikácií.

Kapitola 6

Testovanie GUI aplikácií

Funkčnosť popísaného systému pre automatické testovanie GUI aplikácií som experimentálne overil na aplikáciach *gedit*[1] a *Calculator*[2]. V tejto kapitole je uvedený postup inštalácie a prípravy systému na testovanie GUI a popis vykonaných testov. Vo výsledkoch testovania sa používajú kritéria pokrytia, definované v časti 4.5.

Súbory obsahujúce zdrojový kód systému pre automatizované testovanie, sú spolu so súbormi použitými pri uvedenom testovaní uložené na priloženom CD nosiči. Štruktúra jeho obsahu je popísaná v prílohe A. Uvedené testy používajú ako parametre testovaných aplikácií textové súbory, ktoré sú v popise testovacej sady špecifikované relatívnymi cestami. Preto je pri používaní systému potrebné zachovať štruktúru adresárov rovnakú ako na priloženom CD, alebo upraviť relatívne cesty k týmto súborom.

6.1 Príprava systému

Systém pre automatizované testovanie GUI bol testovaný pod 64-bitovou verziou operačného systému Ubuntu 12.04, s jadrom verzie 3.2.0 a prostredím Unity, ktoré je postavené na prostredí GNOME 3.2.1. Pri ďalšom popise sa preto predpokladá, že na počítači je nainštalovaný práve tento systém s aktualizovanými softvérovými balíkmi. Balíky uvedené v tejto časti, ktoré je potrebné nainštalovať, sú dostupné prostredníctvom manažéra balíkov, a sú priložené aj na CD nosiči v adresári: `packages`.

Prvým krokom je povolenie používania asistenčných technológií v rámci prostredia GNOME. To je možné jednoducho zabezpečiť napríklad spustením aplikácie *Orca*, ktorá číta obsah obrazovky. Aplikácia sa spúšťa prostredníctvom nástroja *System settings*, kde sa v časti *Accessibility* spustí program *Screen reader*. Jeho inštanciu je možné hneď po spustení ukončiť, nakoľko ostatné aplikácie už budú podporovať asistenčné technológie.

Ďalej je potrebné nainštalovať systém LDTP verzie 3.0.0. Archív s jeho zdrojovým kódom je k dispozícii na adrese `http://ldtp.freedesktop.org/wiki/Download/`. Obsah stiahnutého archívu sa extrahuje a podľa pokynov v súbore `README` sa systém preloží a nainštaluje. Pre preklad tohto systému je potrebné nainštalovať softvérové balíky:

- `python-pyatspi2`,
- `python-twisted-web`,
- `python-wnck` a

[1] Aplikácia gedit je oficiálny textový editor prostredia GNOME. [23]
[2] Calculator je predvolená kalkulačka prostredia GNOME. [22]

- `python-gnome2`.

Po ich inštalácií sa systém LDTP preloží a nainštaluje príkazmi:

```
$ python setup.py build
$ sudo python setup.py install
```

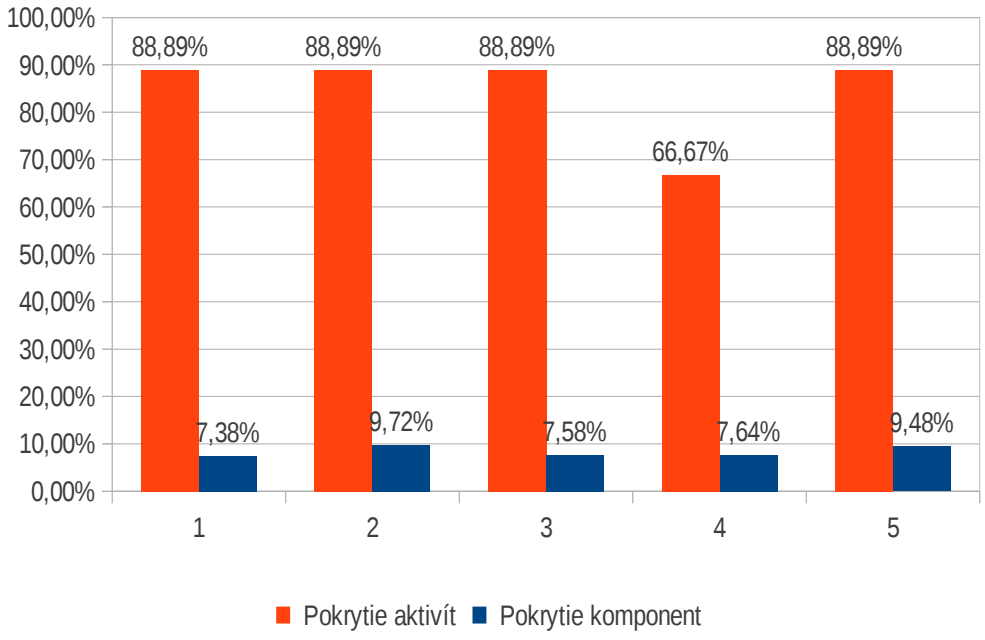
Nasleduje príprava systému pre automatizované testovanie GUI, ktorého zdrojové kódy sú priložené k tejto práci na CD nosiči. Pre preklad systému je potrebné nainštalovať softvérové balíky:

- `bison-2.5`,
- `flex-2.5`,
- `g++-4.6`,
- `libtinyxml2.6.2` a
- `libtinyxml-dev`.

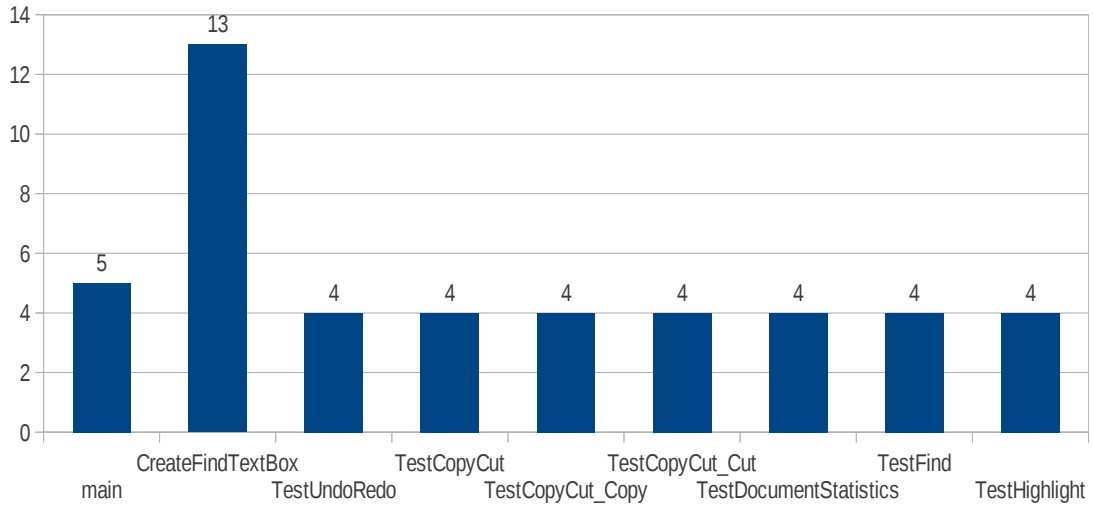
Po ich inštalácií sa systém preloží príkazom `make`. Činnosť systému je implementovaná v preloženom programe `gui-ldtp-tester`.

6.2 Testovanie aplikácie gedit

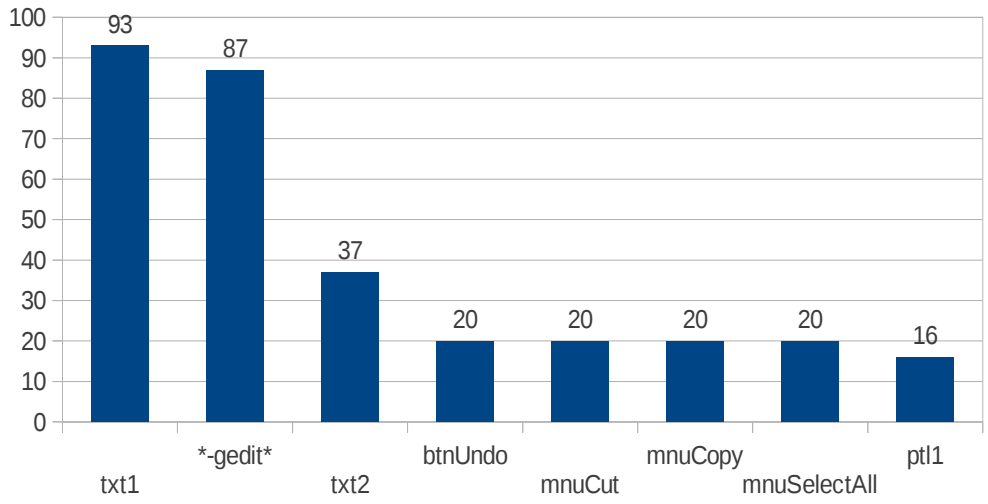
Pre testovanie funkčnosti GUI aplikácie gedit som pripravil testovaciu sadu, ktorej popis je uvedený v prílohe C. Táto testovacia sada otestuje niekoľko funkcií aplikácie gedit nasledovným spôsobom.



Obr. 6.1: Hodnoty pokrytia jednotlivých testovacích behov



Obr. 6.2: Počet vykonávania aktivít v rámci testovacej sady



Obr. 6.3: Najviac testované komponenty v testovacej sade

Aplikácii sa najskôr nastaví premenná prostredia `LANG`, ktorá zaistí načítanie anglickej lokalizácie. Potom je testovaná aplikácia spustená s parametrom určujúcim testovací súbor, ktorý bude po štarte načítaný. Tento súbor obsahuje testovací text, ktorého zobrazenie sa overí po spustení aplikácie. Ďalej je potrebné zobraziť panel pre vyhľadávanie textu, čím je zabezpečené priradenie identifikátoru `txt2`, k tomuto objektu typu TextBox. Teraz má objekt typu TextBox zobrazujúci obsah testovacieho súboru identifikátor `txt1` a TextBox pre vyhľadávanie textu má identifikátor `txt2`. V prípade otvorenia ďalších súborov bude ich obsah zobrazený v objektoch typu TextBox s identifikátormi `txt3`, `txt4`, ...

Potom nasleduje vykonanie tela jednotlivých testovacích prípadov. Každý testovací prípad bude zložený zo sekvencie volaní štyroch aktivít. Testovacia sada obsahuje celkom 5 aktivít, ktoré testujú nasledovnú funkcionalitu.

- `TestUndoRedo` – testuje správnu odozvu GUI pri použití funkcie *undo* a *redo*, ktoré umožňujú vrátiť poslednú vykonanú akciu.
- `TestCopyCut` – testuje správnu odozvu GUI pri kopírovaní a presúvaní obsahu jedného súboru, do nového súboru.
- `TestDocumentStatistics` – overuje, či dialóg zobrazujúci štatistické údaje o obsahu otvoreného súboru zobrazuje správne údaje.
- `TestFind` – testuje správnu funkčnosť komponenty pre vyhľadávanie textu v obsahu súboru.
- `TestHighlight` – testuje zobrazenie obsahu súboru, zvýrazneného ako zdrojový kód rôznych programovacích jazykov.

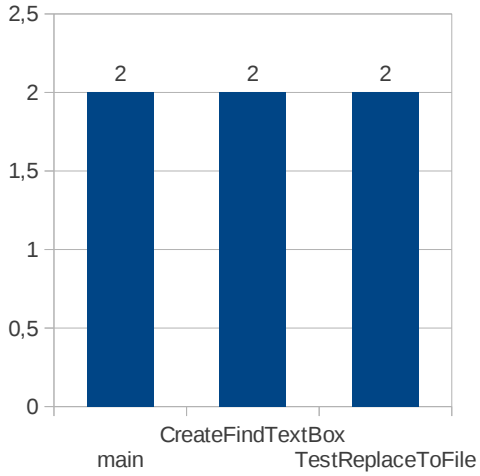
Uvedené aktivity testujú funkcionalitu tak, aby po ich vykonaní bolo GUI v rovnakom stave ako na začiatku aktivity. Tým je možné tieto aktivity vykonávať v rôznych kombináciách. V rámci testovacej sady sa vygeneruje 5 testovacích prípadov a vykoná sa 5 testovacích behov. Súbory obsahujúce výstupy testovania sú uložené na CD nosiči v adresári `gui-ldtp-tester/experiments/gedit`.

Na obrázku 6.1 je graf znázorňujúci hodnoty pokrytia podľa definovaných kritérií jednotlivých testovacích behov. Tento graf ukazuje, že v každom testovacom behu boli otestované iba 4 čiastkové testy z 5-tich definovaných. V testovacom behu č. 4 sa neotestovala funkcionalita pozostávajúca z vykonania viacerích aktivít. V rámci celej testovacej sady bolo definovaných 9 aktivít a všetky boli vykonané. Pokrytie aktivít má preto hodnotu 100 %. Počet vykonaní jednotlivých aktivít v rámci celej testovacej sady je uvedený na obrázku 6.2.

Počas vykonávania testovacej sady bolo identifikovaných 422 komponent GUI a 380 z nich nebolo v testoch vôbec použitých. Pokrytie komponent má preto hodnotu 9,95 %. Hodnota pokrytia komponent závisí hlavne na vytvorených čiastkových testoch a schopnosti testera zahrnúť do týchto testov čo najviac komponent GUI. Na obrázku 6.3 sú uvedené komponenty, ktoré boli v testovacej sade najviac používané. Komponenta s identifikátorom `txt1` slúži na zobrazovanie obsahu testovacieho súboru, komponenta `*-gedit*` reprezentuje hlavné okno aplikácie a komponenta `txt2` slúži na zadávanie textového reťazca vyhľadávaného v súbore.

6.3 Testovanie obsahu súboru – aplikácia gedit

Ďalšie testovanie aplikácie gedit prezentuje možnosti komplexnejšieho overovania funkcionality príkazom `embedded-test` a generovanie testovacej sady použitím generovacieho



Obr. 6.4: Počet vykonávania aktivít v rámci testovacej sady

kritéria `random-data-coverage` s limitom `*`. Popis testovacej sady, v prílohe C, otestuje nahradzovanie častí obsahu testovacieho súboru.

Testovacia sada je tvorená jedným testovacím prípadom, ktorý však bude vykonávaný v rámci niekoľkých testovacích behov. V každom testovacom behu otvorí gedit po svojom spustení náhodný testovací súbor a otestuje sa zobrazenie jeho obsahu v aplikácií. Potom sa vykoná telo testovacieho prípadu a aplikácia sa ukončí. Gedit bude spustený vždy s anglickou lokalizáciou.

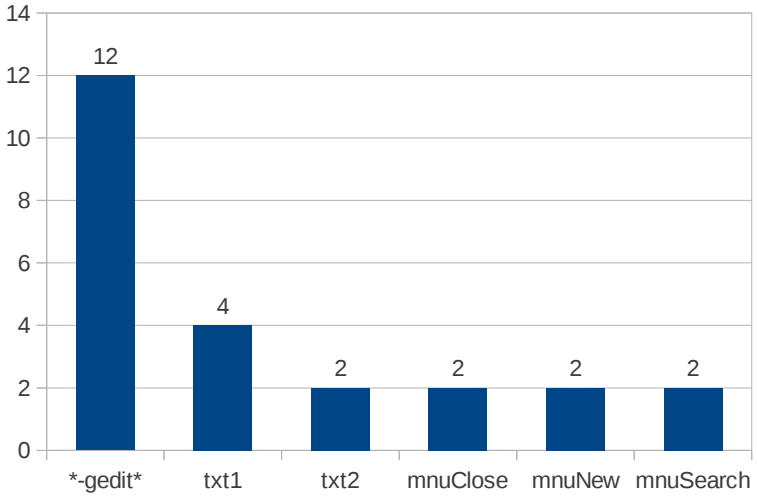
Testovací prípad najskôr pomocou príkazu `embedded-test` zaistí, aby v domovskom adresári aktuálneho užívateľa neexistoval súbor `created.txt`. Aplikácia gedit potom načíta testovací súbor a jeho obsah skopíruje do nového súboru. V novom súbore nahradí všetky výskyty slová `fault` slovom `success` a tento súbor uloží pod názvom `created.txt` do domovského adresára aktuálneho užívateľa. Prostredníctvom príkazu `embedded-test` sa skontroluje či nový súbor, uložený na disku, neobsahuje žiaden výskyt slova `fault`. Na záver sa vytvorený súbor zmaže.

Počas testovania sa postupne vykonávajú nové testovacie behy, až pokým systém neobdrží signál na ukončenie. Vtedy sa dokončí aktuálny testovací beh a testovanie skončí. V tomto prípade boli vykonané 2 testovacie behy s použitím rôznych testovacích súborov, pričom v každom z behov bol vykonaný rovnaký testovací prípad. Výstupy testovania sú uložené na CD nosiči v adresári `gui-ldtp-tester/experiments/gedit-find-replace`.

Keďže testovacia sada obsahovala iba jeden testovací prípad, pokrytie aktivít malo pre každý testovací beh rovnakú hodnotu: 100 %. Pokrytie komponent malo tiež vo všetkých testovacích behoch rovnakú hodnotu: 3,90 %. Keďže v tomto prípade bola testovaná iba jedna vlastnosť aplikácie gedit, bolo toto pokrytie menšie ako v prípade pokrytia komponent predchádzajúceho testu, ktorý je uvedený v časti 6.2.

V rámci celej testovacej sady boli definované 3 aktivity a všetky boli vykonané. Pokrytie aktivít malo v rámci testovacej sady hodnotu 100 %. Aktivity a počet ich vykonaní v testovacej sade je uvedený na obrázku 6.4.

Počas vykonávania testovacej sady bolo identifikovaných 489 komponent a 470 z nich nebolo pri testovaní vôbec použitých. Pokrytie komponent malo hodnotu 3,89 %. Najčas-



Obr. 6.5: Najviac testované komponenty v testovacej sade

tejšie testované komponenty sú uvedené na obrázku 6.5. Komponenta s identifikátorom `*-gedit*` reprezentuje hlavné okno aplikácie a komponenta `txt1` slúži na zobrazovanie obsahu testovacieho súboru.

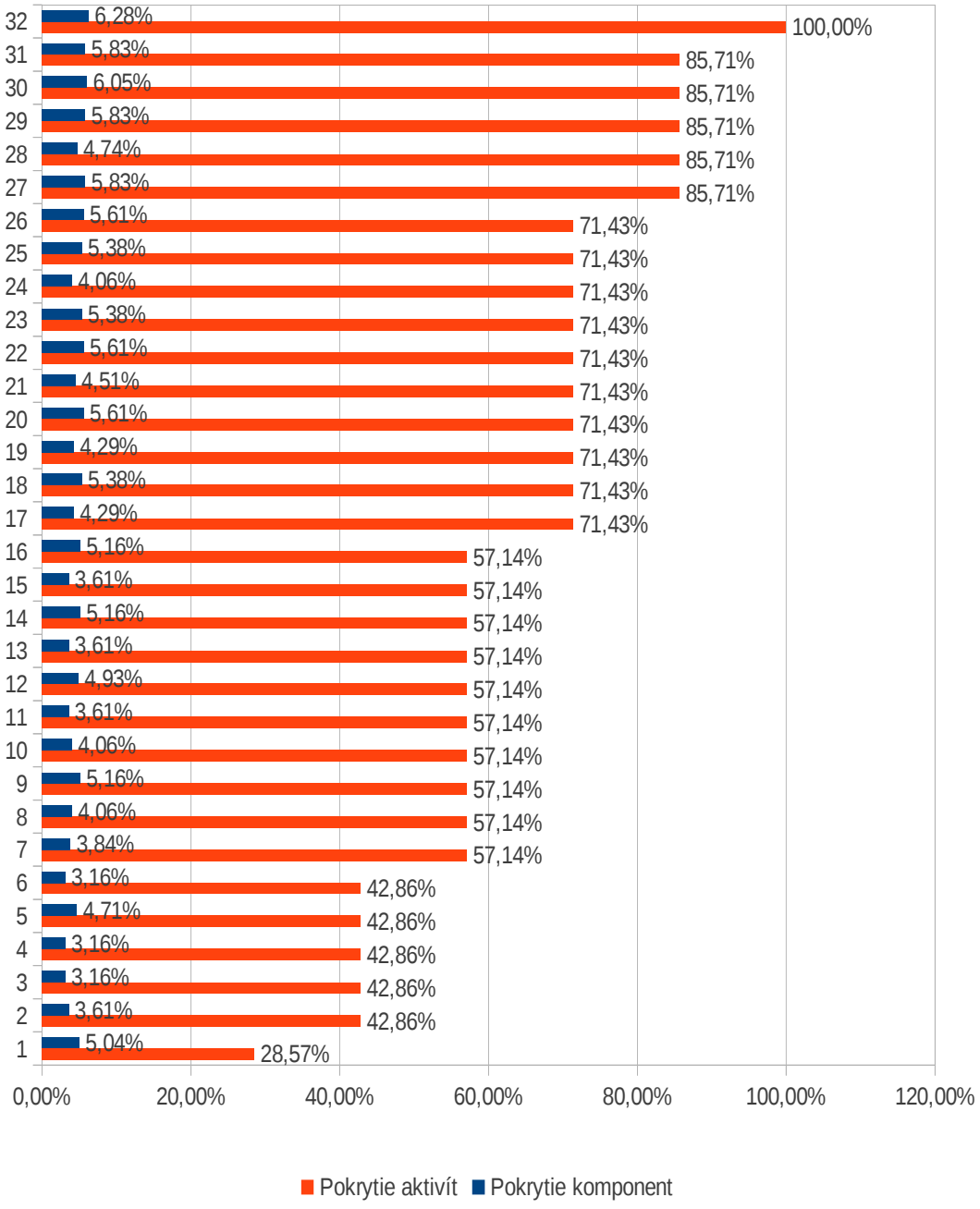
6.4 Testovanie aplikácie Calculator

Pre testovanie aplikácie Calculator som pripravil testovaciu sadu, ktorej popis je uvedený v prílohe C. V rámci tejto testovacej sady sa vykoná niekoľko nasledujúcich aktivít.

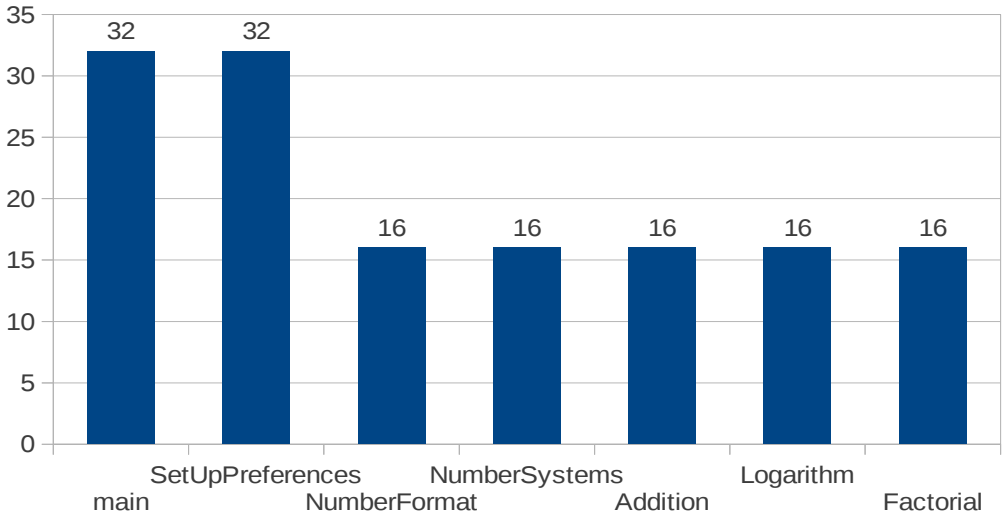
- `Addition` – testuje správnu odozvu GUI pri výpočte súčtu dvoch čísel.
- `Logarithm` – testuje správnu odozvu GUI pri výpočte logaritmu.
- `Factorial` – testuje správnu odozvu GUI pri výpočte faktoriálu.
- `NumberSystems` – testuje správnu odozvu GUI pri prevode čísla do podporovaných číselných sústav.
- `NumberFormat` – testuje správnu odozvu GUI pri zobrazení čísla v rôznom formáte.
- `SetUpPreferences` – nastaví základný formát zobrazenia čísiel pred každým testovaním.

Počas generovania testovacej sady sa vygenerujú sekvencie volania týchto aktivít. Keďže v popise testovacej sady je ako limit kritéria `sequence-criterion` zadaný znak `*`, testovacia sada bude obsahovať všetky sekvencie o dĺžke 0, 1, 2, 3, 4 a 5.

V rámci testovacej sady sa vygenerovalo 32 testovacích prípadov, ktoré boli postupne spustené v 32 testovacích behoch. Výstupy testovania sú uložené na CD nosiči v adresári `gui-ldtp-tester/experiments/calculator`. Pokrytie aktivít a komponent pre jednotlivé testovacie behy je uvedené v grafe na obrázku 6.6. Pokrytie aktivít stúpa spolu s dĺžkou



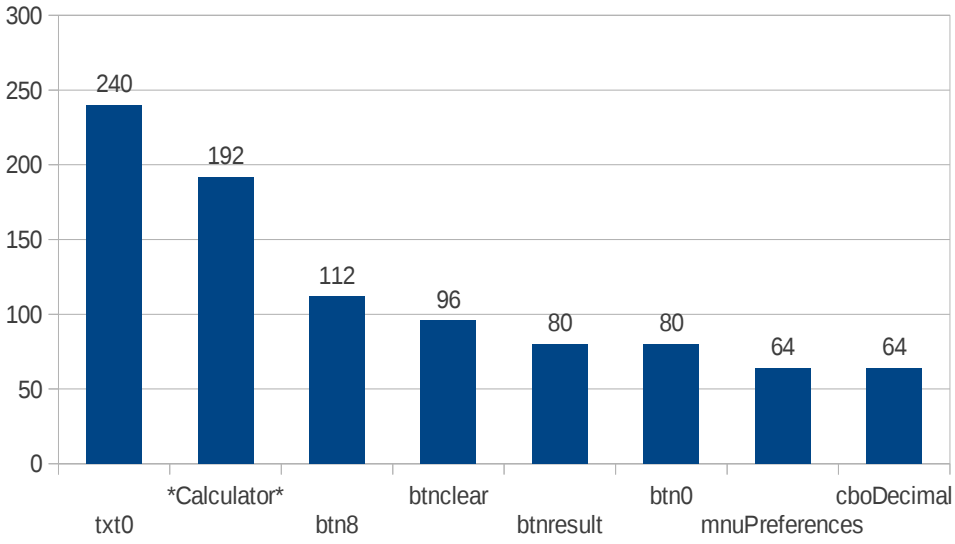
Obr. 6.6: Hodnoty pokrytia jednotlivých testovacích behov



Obr. 6.7: Počet vykonávania aktivít v rámci testovacej sady

sekvencií udalostí vygenerovaných pre daný testovací prípad, no pokrytie komponent stúpa s dĺžkou sekvencií iba mierne.

Popis testovacej sady obsahoval 7 aktivít, z ktorých bola každá vykonaná aspoň raz. Pokrytie aktivít v rámci celej testovacej sady malo hodnotu 100 %. Počet vykonania jednotlivých aktivít je uvedený na obrázku 6.7, kde je viditeľné postupné generovanie čoraz dlhších sekvencií udalostí.



Obr. 6.8: Najviac testované komponenty v testovacej sade

Počas vykonávania testovacej sady bolo identifikovaných 446 komponent a 418 z nich nebolo pri testovaní vôbec použitých. Pokrytie komponent malo hodnotu 6,28 %. Najviac testované komponenty sú uvedené na obrázku 6.8. Komponenta s identifikátorom `txt0` je používaná na zápis matematickej operácie a zobrazenie výsledku. Komponenta `*Calculator*` reprezentuje hlavné okno aplikácie.

6.5 Chyby objavené testovaním

Počas vykonávania uvedených testov bola odhalená chybná funkčnosť služby prostredia Unity, v ktorom boli vykonávané testy. Jednalo sa o program `unity-panel-service`, ktorý však nebol predmetom testovania.

Na základe informácií poskytnutých programom *Apport*[3] som zistil, že uvedená služba `unity-panel-service` obdržala počas vykonávania testov signál `SIGSEGV`, a následne bola ukončená. Predpokladám, že tento problém je spôsobený chybou v podpore asistenčných technológií uvedenej služby. Táto chyba bola ihneď po objavení reportovaná prostredníctvom programu Apport.

[3]Program pre reportovanie chýb v rámci operačného systému Ubuntu.

Kapitola 7

Záver

V rámci tejto práce som naštudoval informácie o testovaní GUI a popísal niekoľko rôznych spôsobov testovania. Ďalej som sa oboznámil s používaním asistenčných technológií a so systémom LDTP pre testovanie GUI.

Na základe získaných znalostí som navrhol systém pre automatizované testovanie GUI a jazyk pre popis testovacej sady, ktorá je týmto systémom generovaná. Tento jazyk umožňuje definovať čiastkové testy jednotlivých funkcií aplikácie v podobe aktivít a popísať spôsob generovania sekvencií týchto testov do výslednej testovacej sady. Okrem toho je ním možné popísať vykonávanie vygenerovaných testovacích prípadov v rôznom prostredí a s rôznymi parametrami testovanej aplikácie. Príkazy uvedeného jazyka sa delia na udalosti umožňujúce manipuláciu s komponentami testovaného GUI, operátory slúžiace pre získavanie informácií z komponent a riadiace príkazy špecifikujúce vykonávanie výsledných testovacích prípadov. Prostredníctvom tohto jazyka je možné používať podmnožinu príkazov systému LDTP.

Navrhnutý systém pre automatizované testovanie GUI som následne implementoval. Tento systém preloží zadaný popis testovacej sady a na jeho základe vygeneruje výslednú testovaciu sadu. Vygenerované testovacie prípady následne vykoná, a tak otestuje požadovanú funkčnosť GUI testovanej aplikácie. K testovaným komponentám GUI pristupuje prostredníctvom systému LDTP, ktorý využíva asistenčné technológie. Počas vykonávania testovacej sady zaznamenáva systém údaje o vykonanom testovaní, čím generuje výsledky a pokrytie testovania.

Pre vyhodnotenie pokrytia vykonaných testov som navrhol dve kritériá: pokrytie aktivít a pokrytie komponent. Pokrytie aktivít poskytuje informáciu o tom, koľko percent z čiastkových testov, definovaných v rámci popisu testovacej sady, bolo počas testovania vykonaných. Pokrytie komponent zase informuje, koľko percent komponent GUI aplikácie, detekovaných počas testovania, bolo pri testovaní použitých.

Funkčnosť implementovaného systému som prezentoval otestovaním GUI vybraných aplikácií. Vytvoril som 3 popisy testovacích sád, ktoré ukazujú možnosti vytvárania testovacej sady prostredníctvom navrhnutého jazyka. V uvedených popisoch som definoval niekoľko čiastkových testov, ktoré testujú jednotlivé funkcie aplikácie gedit alebo Calculator. Výsledky vykonaného testovania som uviedol v tejto práci spolu s vyhodnotením ich pokrytia podľa definovaných kritérií.

Z výsledkov testovania vyplýva, že hodnota pokrytia komponent závisí hlavne na vytvorených čiastkových testoch a schopnosti užívateľa systému zahrnúť do týchto testov čo najviac komponent GUI. Toto kritérium poskytuje informácie o tom, aká veľká časť GUI bola testovaním pokrytá. Hodnota pokrytia aktivít zase závisí na spôsobe generovania

testovacej sady. Generovanie dlhších sekvencií čiastkových testov má za následok vyššiu hodnotu pokrytia aktivít. Uvedené kritérium poskytuje informáciu o tom, aký veľký objem funkcionality GUI bol testovaním pokrytý.

Testovanie GUI týmto systémom neodhalilo chybu v žiadnej z uvedených aplikácií. Napriek tomu sa mi však podarilo nájsť chybu v službe prostredia, v ktorom bolo toto testovanie vykonané. Objavenú chybu som hneď po jej výskyte reportoval.

Táto práca ponúka niekoľko možností jej ďalšieho rozšírenia. Jedným z nich je zahrnutie ďalších príkazov, dostupných v systéme LDTP, do uvedeného systému pre automatizované testovanie a jeho jazyka. Systém som navrhol tak, aby bolo rozšírenie množiny jeho príkazov jednoduché. Taktiež je možné vytvoriť nové riadiace príkazy, ktoré budú poskytovať ďalšie možnosti pri vykonávaní testovacích prípadov. Okrem toho je možné navrhnúť a implementovať nové, zložitejšie kritériá generovania testovacej sady. Na základe tejto práce som tiež vytvoril článok, ktorý bol publikovaný v zborníku konferencie STUDENT EEICT 2013.

Literatúra

[1] AccessibleTech.org: How does accessibility differ across operating systems? [online], [cit. 2013-03-21].
URL http://www.accessibletech.org/access_articles/os/osAccess.php

[2] AccessibleTech.org: What is Accessible Electronic and Information Technology? [online], [cit. 2013-03-21].
URL http://www.accessibletech.org/access_articles/general/whatIsAccessibleEIT.php

[3] AccessibleTech.org: What is Assistive Technology? [online], [cit. 2013-03-20].
URL http://www.accessibletech.org/assist_articles/general/whatAT.php

[4] AccessibleTech.org: What is universal design? [online], [cit. 2013-03-20].
URL http://www.accessibletech.org/access_articles/general/universalDesign.php.

[5] AccessibleTech.org: What makes electronic and information technology inaccessible to people with disabilities? [online], [cit. 2013-03-20].
URL http://www.accessibletech.org/access_articles/general/EITinaccessible.php.

[6] Alagappan, N.: Home – Linux Desktop Testing Project. [online], poslední modifikace: 14. února 2013. [cit. 2013-03-21].
URL http://ldtp.freedesktop.org/wiki/Home

[7] Alagappan, N.: Linux Desktop Testing Project – LDTP. [online], [cit. 2013-03-21].
URL http://download.freedesktop.org/ldtp/doc/ldtp-tutorial.pdf

[8] Ammann, P.; Offutt, J.: *Introduction to Software Testing*. Cambridge University Press, 2008, iSBN 978-0-521-88038-1.

[9] Chang, T.-H.; Yeh, T.; Miller, R. C.: GUI testing using computer vision. In *CHI '10 Proceedings of the 28th international conference on Human factors in computing systems*, New York, NY, USA: Association for Computing Machinery, Inc., 2010, ISBN 978-1-60558-929-9, s. 1535–1544.

[10] Digia: QtTestLib 5.0: Qt Test Overview. [online], [cit. 2013-03-20].
URL https://qt-project.org/doc/qt-5.0/qttestlib/qtest-overview.html

[11] Free Software Foundation, Inc.: Xnee – GNU Project. [online], poslední modifikace: 1. ledna 2010. [cit. 2013-03-20].
URL http://www.gnu.org/software/xnee/

[12] Memon, A.: GUI Testing: Pitfalls and Process. *IEEE Computer*, 08 2002.

[13] Memon, A. M.; Soffa, M. L.; Pollack, M. E.: Coverage Criteria for GUI Testing. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA: Association for Computing Machinery, Inc., 2001, ISBN 1-58113-390-1, s. 256–267.

[14] Sandklef, H.: GNU Xnee. [online], [cit. 2013-03-20].
URL http://www.sandklef.com/xnee/

[15] Sandklef, H.: Testing Applications with Xnee. [online], poslední modifikace: 1. ledna 2004. [cit. 2013-03-20].
URL http://www.linuxjournal.com/article/6660

[16] Sikuli Doc Team: General Information About Sikuli Extensions. [online], poslední modifikace: 8. listopadu 2012. [cit. 2013-03-20].
URL http://doc.sikuli.org/extensions/extensions_info.html

[17] Sikuli Doc Team: How Sikuli Works. [online], poslední modifikace: 8. listopadu 2012. [cit. 2013-03-20].
URL http://doc.sikuli.org/devs/system-design.html

[18] Sikuli Doc Team: Sikuli Documentation. [online], poslední modifikace: 8. listopadu 2012. [cit. 2013-03-20].
URL http://doc.sikuli.org/

[19] Sikuli Doc Team: Sikuli Guide. [online], poslední modifikace: 8. listopadu 2012. [cit. 2013-03-20].
URL http://doc.sikuli.org/extensions/sikuli-guide/index.html

[20] Sikuli Doc Team: Sikuli Script. [online], [cit. 2013-03-20].
URL http://www.sikuli.org/

[21] Sikuli Doc Team: Tutorials. [online], poslední modifikace: 8. listopadu 2012. [cit. 2013-03-20].
URL http://doc.sikuli.org/tutorials/index.html

[22] The GNOME Project: Calculator – GNOME Live! [online], poslední modifikace: 14. října 2012. [cit. 2013-04-03].
URL https://live.gnome.org/Calculator

[23] The GNOME Project: gedit. [online], poslední modifikace: 3. dubna 2013. [cit. 2013-04-03].
URL http://projects.gnome.org/gedit/

[24] The GNOME Project: How Accessibility Works in GNOME. [online], [cit. 2013-03-21].
URL https://developer.gnome.org/accessibility-devel-guide/stable/gad-how-it-works.html.en

[25] The GNOME Project: Universal access. [online], [cit. 2013-03-21].
URL https://help.gnome.org/users/gnome-help/stable/a11y.html

[26] The Jython Project: Jython: General Information. [online], poslední modifikace: 3. března 2012. [cit. 2013-03-20].
URL http://wiki.python.org/jython/JythonFaq/GeneralInfo

[27] W3C: WCAG Overview. [online], [cit. 2013-03-20].
URL http://www.w3.org/WAI/intro/wcag.php

[28] Xie, Q.: Developing Cost-Effective Model-Based Techniques for GUI Testing. In *Proceeding ICSE '06: Proceedings of the 28th international conference on Software engineering*, New York: ACM, 2006, ISBN 1-59593-375-1, s. 997–1000.

[29] Yeh, T.; Chang, T.-H.; Miller, R. C.: Sikuli: Using GUI Screenshots for Search and Automation. In *UIST '09 – Proceedings of the 22th annual ACM Symposium on User Interface Software and Technology*, Victoria, BC, Canada: Association for Computing Machinery, Inc., 2009, s. 183–192.

Dodatok A

Obsah priloženého CD

K tejto práci je priložený CD nosič, ktorý obsahuje zdrojové kódy systému popísaného v kapitolách 4 a 5 , a tiež súbory použité pri testovaní popísanom v kapitole 6. Jeho adresárová štruktúra je nasledovná.

- `gui-ldtp-tester/src` obsahuje zdrojové kódy systému pre automatizované testovanie GUI.
- `gui-ldtp-tester/experiments/gedit` je adresár testovania aplikácie gedit, ktorý obsahuje popis testovacej sady, súbory s testovacím textom a adresáre pre vygenerovanú testovaciu sadu a pre výstupy testovania.
- `gui-ldtp-tester/experiments/gedit-find-replace` je adresár testovania práce s obsahom súboru v aplikácii gedit, ktorý obsahuje popis testovacej sady, súbor s testovacím textom a adresáre pre vygenerovanú testovaciu sadu a pre výstupy testovania.
- `gui-ldtp-tester/experiments/calculator` je adresár testovania aplikácie Calculator, ktorý obsahuje popis testovacej sady a adresáre pre vygenerovanú testovaciu sadu a pre výstupy testovania.
- `packages` je adresár, ktorý obsahuje balíky potrebné k prekladu a behu programu `gui-ldtp-tester`.
- `tex` je adresár, ktorý obsahuje zdrojové súbory tohto dokumentu.

Jednotlivé testovania obsahujú adresár `generated`, v ktorom je testovacia sada vygenerovaná na základe príslušného popisu. V adresároch `results` sú uložené výstupy jednotlivých testovaní.

Dodatok B

Zoznam príkazov jazyka

V časti 4.1 je uvedený návrh jazyka pre popis testovacích sád. Táto príloha obsahuje zoznam udalostí emulujúcich akcie užívateľa a operátorov, získavajúcich informácie z GUI.

B.1 Udalosti pre emuláciu akcií užívateľa

V jazyku pre popis testovacej sady je možné použiť nasledovné udalosti:

- `<event> ::= "assert" <expression> <string>`
  Príklad: `assert (editable "txt0") "Text box should be editable"`
  Popis: Ak komponenta `txt0` nie je editovateľná, bola počas testovania objavená chyba a testovanie končí. Text správy sa zapíše do výstupného súboru.

- `<event> ::= "click" <string>`
  Príklad: `click "btnOK"`
  Popis: Vykoná sa kliknutie ľavým tlačidlom myši na komponentu `btnOK`.

- `<event> ::= "do" <identifier>`
  Príklad: `do TestMenu`
  Popis: Vykoná sa aktivita s identifikátorom `TestMenu`.

- `<event> ::= "double-click" <string>`
  Príklad: `double-click "btnOK"`
  Popis: Vykoná sa dvojité kliknutie ľavým tlačidlom myši na komponentu `btnOK`.

- `<event> ::= <string> "double-click-row" <string>`
  Príklad: `"tblFiles" double-click-row "file.txt"`
  Popis: V tabuľke `tblFiles` sa vykoná dvojité kliknutie ľavým tlačidlom myši na riadok, ktorý v prvom stĺpci obsahuje text: `file.txt`.

- `<event> ::= "embedded-test" ľubovoľný_text "embedded-test-end"`
  Príklad:
  ```
  embedded-test
  print 'Hello world'
  embedded-test-end
  ```
  Popis: Text tohto príkazu je interpretovaný ako kód jazyka Python. Konkrétny príkaz vypíše na výstup text: `Hello world`.

- `<event> ::= "grab-focus" <string>`
  Príklad: `grab-focus "txt1"`
  Popis: Komponente `txt1` sa nastaví zameranie klávesnice.

- `<event> ::= "check" <string>`
  Príklad: `check "chkShowLineNumbers"`
  Popis: Komponenta `chkShowLineNumbers` typu *CheckBox* sa označí.

- `<event> ::= "keyboard-event" <string>`
  Príklad: `keyboard-event "<alt>s"`
  Popis: Vygenerujú sa udalosti stlačenia kláves `Alt` a `s`. V rámci tejto udalosti je možné generovať stlačenie riadiacich kláves reťazcami: `<ctrl>`, `<enter>`, `<up>`, `<f1>`, ...

- `<event> ::= "on" <string>`
  Príklad: `on "dlgOpenFile"`
  Popis: Aktuálny kontext vykonávania príkazov sa nastaví na dialóg `dlgOpenFile` a skontroluje sa existencia tohto dialógu.

- `<event> ::= <string> "select" <string>`
  Príklad: `"cboFormat" select "mnuBinary"`
  Popis: V komponente `cboFormat` typu *ComboBox* sa vyberie položka `mnuBinary`.

- `<event> ::= <string> "select-row-contains" <string>`
  Príklad: `"tblFiles" select-row-contains "file.txt"`
  Popis: V tabuľke `tblFiles` sa označí riadok, ktorý v prvom stĺpci obsahuje text: `file.txt`.

- `<event> ::= <string> "select-tab" <string>`
  Príklad: `"ptl1" select-tab "*Untitled*"`
  Popis: Zobrazí sa komponenta typu *PageTab*, ktorej popisok obsahuje text: `Untitled`, a ktorá je v komponente `ptl1` typu *PageTabList*.

- `<event> ::= <string> "set-text-value" <string>`
  Príklad: `"txt0" set-text-value "HelloWorld"`
  Popis: Do komponenty `txt0` sa vpíše text: `HelloWorld`.

- `<event> ::= <string> "set-value" <int>`
  Príklad: `"sbtnDecimalPlaces" set-value 2`
  Popis: Komponente `sbtnDecimalPlaces` typu *SpinButton* sa nastaví hodnota `2`.

- `<event> ::= "uncheck" <string>`
  Príklad: `uncheck "chkShowLineNumbers"`
  Popis: Komponente `chkShowLineNumbers` typu *CheckBox* sa zruší označenie.

- `<event> ::= "unselect-text" <string>`
  Príklad: `unselect-text "txt1"`
  Popis: Zruší sa označenie textu v komponente `txt1`.

- `<event> ::= "wait" <int-not-negative>`
  Príklad: `wait 2`
  Popis: Vykonávanie testovania bude pozastavené na dobu 2 sekundy.

- `<event> ::= "window-close" <string>`
  Príklad: `window-close "*Calculator*"`
  Popis: Okno, ktorého titulok obsahuje text: `Calculator`, sa zatvorí. V prípade, že je titulok okna špecifikovaný prázdnym reťazcom alebo znakom `*`, zatvoria sa okná všetkých spustených aplikácií.

B.2 Operátory pre získavanie informácií

Ďalšou skupinou sú operátory slúžiace na získanie informácií o aktuálnom stave komponent GUI.

- `<operation> ::= "default" <string>`
  Návratová hodnota: `<bool>`
  Príklad: `default "btnOK"`
  Popis: Zistí, či je tlačidlo `btnOK` nastavené ako predvolené.

- `<operation> ::= "editable" <string>`
  Návratová hodnota: `<bool>`
  Príklad: `editable "txtName"`
  Popis: Zistí, či je komponenta `txtName` editovateľná.

- `<operation> ::= "enabled" <string>`
  Návratová hodnota: `<bool>`
  Príklad: `enabled "btnOK"`
  Popis: Zistí, či je tlačidlo `btnOK` povolené.

- `<operation> ::= "get-label" <string>`
  Návratová hodnota: `<string>`
  Príklad: `get-label "btnOK"`
  Popis: Vráti hodnotu vlastnosti `label`, komponenty `btnOK`.

- `<operation> ::= "get-row-count" <string>`
  Návratová hodnota: `<int>`
  Príklad: `get-row-count "tblFiles"`
  Popis: Vráti počet riadkov tabuľky `tblFiles`.

- `<operation> ::= "get-tab-count" <string>`
  Návratová hodnota: `<int>`
  Príklad: `get-tab-count "ptl1"`
  Popis: Vráti počet komponent typu *PageTab*, ktoré sú potomkami komponenty `ptl1` typu *PageTabList*.

- `<operation> ::= "get-text-value" <string>`
  Návratová hodnota: `<string>`
  Príklad: `get-text-value "txt1"`
  Popis: Vráti text, ktorý je zobrazený v komponente `txt1`.

- `<operation> ::= "checked" <string>`
  Návratová hodnota: `<bool>`
  Príklad: `checked "chkShowLineNumbers"`
  Popis: Zistí, či je označená komponenta `chkShowLineNumbers`, typu *CheckBox*.

- `<operation> ::= "object-exists" <string>`
  Návratová hodnota: `<bool>`
  Príklad: `object-exists "lblInformation"`
  Popis: Zistí, či v kontexte aktuálneho okna existuje komponenta `lblInformation`.

- `<operation> ::= <string> "row-exists" <string>`
  Návratová hodnota: `<bool>`
  Príklad: `"tblFiles" row-exists "file.txt"`
  Popis: Zistí, či sa v tabuľke `tblFiles` nachádza riadok, ktorého ľubovoľný stĺpec obsahuje text: `file.txt`.

- `<operation> ::= <string> "verify-select" <string>`
  Návratová hodnota: `<bool>`
  Príklad: `"cboFormat" verify-select "mnuBinary"`
  Popis: Zistí, či má komponenta `cboFormat` vybranú položku `mnuBinary`.

- `<operation> ::= "visible" <string>`
  Návratová hodnota: `<bool>`
  Príklad: `visible "btnOK"`
  Popis: Zistí, či je tlačidlo `btnOK` viditeľné.

Dodatok C

Popis testovacích sád

Táto príloha obsahuje popis testovacích sád, ktoré boli použité pri testovaní GUI aplikácií, popísanom v kapitole 6. Tieto popisy sú uložené aj na priloženom CD, v príslušných súboroch. Pre testovanie aplikácie gedit, popísanom v časti 6.2, bol použitý nasledovný popis testovacej sady, ktorý je tiež dostupný na priloženom CD v súbore: `gedit.tsd`.

```
activity main
{
   set-enviroment "LANG" = "en_US.UTF-8"
   run "/usr/bin/gedit ../experiments/gedit/test1.txt"

   on "*-gedit*"

   // check whether some file is opened
   wait 1
   assert (get-text-value "txt1" != "")
      "Content of opened file should be displayed"

   // create textbox "txt2", which will be for text finding
   do CreateFindTextBox

   // tests should pass at every combination
   generate-set sequence-criterion 4
   {
      do TestUndoRedo
      do TestCopyCut
      do TestHighlight
      do TestFind
      do TestDocumentStatistics
   }

   wait 4
   on "*-gedit*"
   click "mnuQuit"
}

/*
 * Show textbox for finding text, wich will create widget txt2.
 */
activity CreateFindTextBox
{
   on "*-gedit*"

   keyboard-event "<ctrl>f"
```

```
   wait 1
   keyboard-event "<escape>"
   wait 1
   assert (object-exists "txt2") "Created txt2 as textbox for finding text"
}

/*
 * Test if undo and redo works properly.
 */
activity TestUndoRedo
{
   on "*-gedit*"

   click "mnuSelectAll"
   click "mnuDelete"

   click "btnUndo"
   wait 2
   assert ((get-text-value "txt1") != "")
      "There should be text after undo of delete action"

   click "btnRedo"
   wait 2
   assert ((get-text-value "txt1") == "")
      "There should not be text after redo of delete action"

   // get content of a file to original state
   click "btnUndo"
   unselect-text "txt1"
}

/*
 * Test if content of an opened file could be copyied to another file.
 */
activity TestCopyCut
{
   on "*-gedit*"

   // create textbox "txt2", which will be for text finding
   do CreateFindTextBox

   // unselect possible selected text
   unselect-text "txt1"
   wait 1

   // check ability of menu items when no text is selected
   assert (not enabled "mnuCut") "Menu Cut should not be enabled"
   assert (not enabled "mnuCopy") "Menu Copy should not be enabled"
   assert (enabled "mnuSelectAll") "Menu Select All should be enabled"

   // prepare new file
   click "btnNew"

   // select content of an opened file
   wait 1
   "ptl1" select-tab "*txt"
   wait 1
   click "mnuEdit"
   click "mnuSelectAll"
```

```
   // check ability of menu items when some text is selected
   wait 2
   assert (enabled ”mnuCut”) ”Menu Cut should be enabled”
   assert (enabled ”mnuCopy”) ”Menu Copy should be enabled”

   // copy or cut text of an opened file
   generate-set event-criterion
   {
      do TestCopyCut_Copy
      do TestCopyCut_Cut
   }

   // paste copyied content to new file
   wait 1
   ”ptl1” select-tab ”*Untitled*”
   wait 1
   click ”mnuPaste”

   wait 2
   assert (get-text-value ”txt3” != ””) ”There should be pasted text”

   // close new file
   click ”btnUndo”
   click ”mnuClose”
   // wait to destroy txt3
   wait 2

   unselect-text ”txt1”
}
activity TestCopyCut_Copy
{
   click ”mnuCopy”
}
activity TestCopyCut_Cut
{
   click ”mnuCut”
   // this content should remains in its original place too
   click ”btnUndo”
}

/*
 * Test whether document statistics dialog is displayed properly.
 */
activity TestDocumentStatistics
{
   on ”*-gedit*”

   // unselect possible selection
   unselect-text ”txt1”

   // invoke document statistics dialog
   click ”mnuTools”
   click ”mnuDocumentStatistics”

   // update statistics information
   on ”*Statistics*”
   click ”btnUpdate”
   wait 2
```

```
   // check that statistics of selected text are 0
   // (there should be 5 label widgets with label 0)
   assert (not (enabled ”lblSelection”))
      ”Label selection should not be enabled”
   assert ((get-label ”lbl0”) == ”0”)
      ”There should be label 0 with label \”0\””
   assert ((get-label ”lbl01”) == ”0”)
      ”There should be label 01 with label \”0\””
   assert ((get-label ”lbl02”) == ”0”)
      ”There should be label 02 with label \”0\””
   assert ((get-label ”lbl03”) == ”0”)
      ”There should be label 03 with label \”0\””
   assert ((get-label ”lbl04”) == ”0”)
      ”There should be label 04 with label \”0\””

   // check that statistics of document are not 0
   // (there should not be more label widgets with label 0)
   assert (enabled ”lblDocument”)
      ”Label Document should not be enabled”
   assert (not (object-exists ”lbl05”))
      ”There should be no more labels containing label \”0\””

   click ”btnClose”
}

/*
 * Test whether it is possible to invoke find widget by menu,
 * by keyboard shortcut, and find text in loaded file.
 */
activity TestFind
{
   on ”*-gedit*”

   // widget for finding can be invoked by menu, or keyboard shortcut
   generate-set event-criterion
   {
      click ”mnuFind”
      keyboard-event ”<ctrl>f”
   }
   wait 1
   assert (visible ”txt2”)
      ”Text box for finding should be visible after <ctrl>f”

   // find text
   ”txt2” set-text-value ”fault”
   keyboard-event ”<enter>”

   wait 1
   assert (get-text-value ”txt1” != ””)
      ”There should be content of file displayed after finding text”

   // clear finding highlighting
   keyboard-event ”<ctrl>f”
   wait 1
   ”txt2” set-text-value ””
   keyboard-event ”<enter>”
   wait 1
   unselect-text ”txt1”
```

```
}

/*
 * Check that content of a file can be highlighted as various document types.
 */
activity TestHighlight
{
   /*
    * There should be opened file containing text.
    */

   on "*-gedit*"

   generate-set event-criterion
   {
      click "mnuHTML"
      click "mnuLaTeX"
      click "mnuXML"
      click "mnuPHP"
      click "mnuPython"
      click "mnuC"
      click "mnuC#"
      click "mnuC++"
      click "mnuMakefile"
      click "mnuPlainText"
   }

   assert (get-text-value "txt1" != "") "Highlighted text should not be empty"
}
```

Popis testovacej sady použitý pri testovaní práce s obsahom súboru v aplikácií gedit, popísanom v časti 6.3, je uložený na priloženom CD v súbore: `find_replace.tsd`. Jeho obsah je nasledovný:

```
activity main
{
   // run application with different files
   generate-set random-data-criterion *
   {
      var TestFiles = {
         "/usr/bin/gedit ../experiments/gedit-find-replace/test1.txt",
         "/usr/bin/gedit ../experiments/gedit-find-replace/test2.txt"}

      set-enviroment "LANG" = "en_US.UTF-8"
      run TestFiles
   }

   on "*-gedit*"

   // check whether some file is opened
   wait 1
   assert (get-text-value "txt1" != "") "File opened"

   wait 1
   do TestReplaceToFile

   wait 4
   on "*-gedit*"
   click "mnuQuit"
```

```
}

/*
 * Show textbox for finding text, wich will create widget txt2.
 */
activity CreateFindTextBox
{
   on "*-gedit*"

   keyboard-event "<ctrl>f"
   wait 1
   keyboard-event "<escape>"
   wait 1
   assert (object-exists "txt2") "Created txt2 as textbox for finding text"
}

/*
 * Test whether replace dialog works properly, regardles of checked
 * combination of its checkboxes.
 */
activity TestReplaceToFile
{
   on "*-gedit*"

   // create textbox "txt2", which will be for text finding
   do CreateFindTextBox

   // delete file ~/created.txt, if it exists
embedded-test
import glob, os
from os.path import expanduser

home_path = expanduser('~')
if os.path.exists(home_path + '/created.txt'):
   old_working_directory = os.getcwd()
   os.chdir(home_path)
   created_file = glob.glob('created.txt')
   os.unlink(created_file[0])
   os.chdir(old_working_directory)
embedded-test-end

   // unselect possible selected text
   unselect-text "txt1"

   // copy content of a file to new file
   click "mnuSelectAll"
   click "mnuCopy"
   click "mnuNew"
   wait 1
   click "mnuPaste"
   wait 1

   click "mnuSearch"
   click "mnuReplace"

   // invoke replace dialog
   on "*Replace*"

   // set text to be replaced and text to replace
```

```
   "txt1" set-text-value "fault"
   wait 1
   assert (enabled "btnFind")
      "Button find should be enabled, because there is text to search for"
   "txt0" set-text-value "success"
   keyboard-event "<alt>a"
   wait 1
   click "btnClose"
   wait 1

   on "*-gedit*"
   click "mnuSaveAs"

   // save new file
   on "*Save*"
   "txtName" set-text-value "~/created.txt"
   wait 1
   keyboard-event "<alt>s"
   wait 3

   // close created file
   on "*-gedit*"
   click "mnuClose"
   wait 1

   // check size of created file
embedded-test
import glob, os
from os.path import expanduser

#load file
home_path = expanduser('~')
test_file = open(home_path + '/created.txt', 'r')
text = test_file.read()
test_file.close()

# check whether file contains "fault"
my_assert(not ('fault' in text), 'Created file should not contains \'fault\'')

# delete created file
old_working_directory = os.getcwd()
os.chdir(home_path)
created_file = glob.glob('created.txt')
os.unlink(created_file[0])
os.chdir(old_working_directory)
embedded-test-end
}
```

V časti 6.4 je popísané testovanie aplikácie Calculator, pri ktorom bol použitý nasledovný popis testovacej sady. Tento popis je uložený aj na priloženom CD v súbore: `calculator.tsd`.

```
activity main
{
   set-enviroment "LANG" = "en_US.UTF-8"
   run "/usr/bin/gcalctool"

   on "*Calculator*"
```

```
   // show calculator in programming mode
   click "mnuProgramming"

   do SetUpPreferences

   generate-set sequence-criterion *
   {
      do Addition
      do Logarithm
      do Factorial
      do NumberSystems
      do NumberFormat
   }

   on "*Calculator*"
   click "mnuQuit"
}

activity SetUpPreferences
{
   on "*Calculator*"

   // set calculator for decimal numbers
   if (not object-exists "cboDecimal")
   {
      if (object-exists "cboBinary")
      {
         "cboBinary" select "mnuDecimal"
      }

      if (object-exists "cboOctal")
      {
         "cboOctal" select "mnuDecimal"
      }

      if (object-exists "cboHexadecimal")
      {
         "cboHexadecimal" select "mnuDecimal"
      }
   }

   wait 1
   click "mnuPreferences"

   on "*Preferences*"
   uncheck "chkShowtrailingzeroes"
   uncheck "chkShowthousandsseparators"
   click "btnClose"
   wait 1
}

activity NumberFormat
{
   on "*Calculator*"

   click "btnclear"
   wait 1
   click "btn8"
   click "btn8"
```

```
   click "btn8"
   click "btn8"
   click "btn8"
   click "btn8"
   click "btn8"
   click "btnresult"

   wait 1
   click "mnuPreferences"

   on "*Preferences*"

   "sbtndecimalplaces" set-value 9
   check "chkShowtrailingzeroes"
   check "chkShowthousandsseparators"
   click "btnClose"

   on "*Calculator*"

   wait 1
   assert (get-text-value "txt0" == "8,888,888.000000000")
      "There should be \'8,888,888.000000000\' in text box, as formatted number"

   click "btnclear"

   click "mnuPreferences"

   on "*Preferences*"

   uncheck "chkShowtrailingzeroes"
   uncheck "chkShowthousandsseparators"
   click "btnClose"
   wait 1
}

activity NumberSystems
{
   on "*Calculator*"

   wait 1
   assert (object-exists "cboDecimal") "Calculator is set to decimal"

   click "btnclear"
   assert (get-text-value "txt0" == "")
   "There should not be any text in text box"

   click "btn1"
   click "btn0"
   click "btn0"
   click "btn0"
   click "btnresult"

   wait 1
   assert (get-text-value "txt0" == "1000")
      "There should be \'1000\' in text box, as decimal number"

   "cboDecimal" select "mnuBinary"
   wait 1
   assert (get-text-value "txt0" == "1111101000")
```

```
      "There should be \'1111101000\' in text box, as binary number"

   "cboBinary" select "mnuOctal"
   wait 1
   assert (get-text-value "txt0" == "1750")
      "There should be \'1750\' in text box, as octal number"

   "cboOctal" select "mnuHexadecimal"
   wait 1
   assert (get-text-value "txt0" == "3E8")
      "There should be \'3E8\' in text box, as hexadecimal number"

   "cboHexadecimal" select "mnuDecimal"
   wait 1
}

activity Addition
{
   on "*Calculator*"
   click "btnclear"
   wait 1
   assert (get-text-value "txt0" == "")
      "There should not be any text in text box"

   click "btn2"
   click "btn0"
   click "btnadd"
   click "btn1"
   click "btn0"

   wait 1
   assert (get-text-value "txt0" == "20+10")
      "There should be \'20+10\' in text box, before addition"

   click "btnresult"

   wait 1
   assert (get-text-value "txt0" == "30")
      "There should be correct result 30 in text box, after addition"
}

activity Logarithm
{
   on "*Calculator*"
   click "btnclear"
   wait 1
   assert (get-text-value "txt0" == "")
      "There should not be any text in text box"

   click "btnlogarithm"
   click "btn1"

   wait 1
   assert (get-text-value "txt0" == "log 1")
      "There should be \'log 1\' in text box, before logarithm"

   click "btnresult"

   wait 1
```

```
   assert (get-text-value ”txt0” == ”0”)
      ”There should be correct result 0 in text box, after logarith”
}

activity Factorial
{
   on ”*Calculator*”
   click ”btnclear”
   wait 1
   assert (get-text-value ”txt0” == ””)
      ”There should not be any text in text box”

   click ”btn6”
   click ”btnfactorial”

   wait 1
   assert (get-text-value ”txt0” == ”6!”)
      ”There should be \’6!\’ in text box, before factorial”

   click ”btnresult”

   wait 1
   assert (get-text-value ”txt0” == ”720”)
      ”There should be correct result 720 in text box, after factorial”
}
```