VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

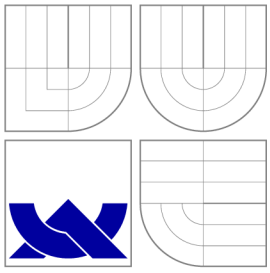
SADA TESTŮ GUI INSTALAČNÍHO NÁSTROJE ANACONDA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

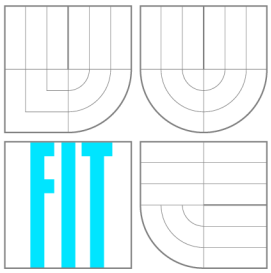
AUTOR PRÁCE
AUTHOR

JAN SEDLÁK

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

SADA TESTŮ GUI INSTALAČNÍHO NÁSTROJE ANACONDA

TESTSUITE OF FRONTEND OF INSTALLATION TOOL ANACONDA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN SEDLÁK
AUTHOR

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

BRNO 2013

Abstrakt

Cílem této práce je prozkoumat principy a dostupné nástroje pro automatizované testování grafického uživatelského rozhraní. Jeden z těchto nástrojů je rozšířen a použit pro návrh a implementaci testů instalace linuxové distribuce Fedora programem Anaconda. Je implementována možnost dynamického generování testovacích případů z menších částí.

Abstract

The aim of this work is to examine principles and tools for automated graphical user interface testing. One of this tools is chosen and modified for implementation of testsuite of Fedora Linux installation using Anaconda. Dynamic generation of test cases from smaller parts is also implemented.

Klíčová slova

automatizované testování, testování GUI, testování programu Anaconda, automatizace instalace, Fedora

Keywords

test automation, GUI testing, Anaconda testing, installation automation, Fedora

Citace

Jan Sedlák: Sada testů GUI instalačního nástroje Anaconda, bakalářská práce, Brno, FIT VUT v Brně, 2013

Sada testů GUI instalačního nástroje Anaconda

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doktora Aleše Smrčky.

.....
Jan Sedlák
15. května 2013

Poděkování

Za poskytnutí odborných konzultací ohledně testování distribuce Fedora bych chtěl poděkovat Kamilu Páralovi a Timu Flinkovi, členům týmu Fedora Quality Assurance a zaměstnancům firmy Red Hat, která vývoj distribuce Fedora zaštiťuje. Za konzultace o detailech programu Anaconda chci poděkovat Vratislavu Podzimkovi, členovi vývojářského týmu programu Anaconda a zaměstnanci firmy Red Hat. Za odborné vedení a cenné připomínky chci poděkovat Aleši Smrčkovi.

Obsah

Kapitola 1

Úvod

Nezávisle na použité metodice zůstává testování důležitou částí vývoje software. Poskytuje zpětnou vazbu o funkčnosti software, informace o kvalitě software a ze své podstaty slouží pro odhalování chyb. Velký význam má regresní testování. I ten nejpropracovanější program se může stát nepoužitelným, pokud se při jeho vývoji nedal na testování dostatečný důraz a výsledný produkt proto obsahuje větší množství nedokonalostí, které uživateli brání v efektivním užití tohoto produktu.

Ve světě komerčního software existují celé specializované skupiny lidí zajišťující kvalitu software, tzv. *Quality Assurance*. Tyto skupiny jsou často složeny z odborníků, kteří se z celého vývoje specializují právě na testování. Používají specializované testovací programy, integrační servery atp. Ve světě otevřeného software je používán spíše přístup popularizovaný Ericem S. Raymondem v jeho práci *Katedrála a tržiště*, nazvaný *Release early, release often*. Tento přístup, ač nevylučuje existenci skupin pro zajištění kvality, nechává testování produktu z větší části na uživatelích. I přes to vznikají projekty s otevřeným zdrojovým kódem, které slouží pro automatizované testování a zajištění kvality software.

V této práci budou představeny principy automatizovaného testování grafického uživatelského rozhraní software a prozkoumány open-source nástroje, které tyto principy implementují. Dále bude jeden z těchto nástrojů použit pro implementaci sady testů pro grafickou instalaci linuxové distribuce Fedora programem Anaconda. V první kapitole bude nastíněna teorie testování, principy testování a jeho dělení. Ve druhé kapitole budou prozkoumány dostupné nástroje s otevřeným kódem, sloužící pro automatizované testování GUI. Třetí kapitola obsahuje návrh sady testů pro testování instalace distribuce Fedora programem Anaconda. Čtvrtá kapitola se zabývá konkrétní implementací testovací sady navržené ve třetí kapitole. V poslední kapitole je zhodnocení pokrytí sady testů a možnosti dalšího rozvoje.

Kapitola 2

Teorie testování

Testování je proces vykonávání software, který má za cíl ověřit, zda software odpovídá specifikovaným požadavkům, a dále odhalit chyby [27]. Slovo *specifikace* v této definici hraje nezanedbatelnou roli. Testování totiž nemůže zaručit absolutní správnost software. Testovat můžeme pouze míru konformity produktu ke specifikaci. Musíme však brát v potaz možnost, že se chyba vyskytla již ve specifikaci programu [20]. S tímto principem úzce souvisí pojmy validace a verifikace.

Validace je zjišťování, do jaké míry softwarový systém odpovídá požadavkům ve smyslu splnění uživatelových potřeb [24].

Verifikace je ověřování souladu implementace software a jeho specifikace [24].

Validace zahrnuje na počátku vývojového cyklu hlubší pochopení zákazníkových požadavků skrze dotazníky či pohovory. Mezi validační testy, které jsou spouštěné ke konci vývojového cyklu, patří systémové testy a testy integrity [17]. Testovat lze však i shodu specifikace a klientových požadavků a návrh produktu. Proces validace zahrnuje funkční testování software. Verifikace je spojená s inspekcí kódu a zkoumáním výstupu programu [11].

2.1 Motivace

Testování pomáhá snížit náklady na výsledný software stejně jak zvýšit jeho kvalitu. Použití testů jednotek, integračních testů či metod vývoje software orientovaných na testy (například *test-driven development*) mohou mít za následek čistší a kvalitnější výsledný kód [10, 15]. Regresní testy zase umožňují snazší refaktorizaci a vedou proto ke stabilnějšímu vývoji v prostředí měnících se požadavků [25]. V knize Testing Computer Software [17] je uveden výzkum který zjistil, že software, který se dostane do fáze testování, obsahuje průměrně tři chyby na sto řádků příkazů. Důvody k testování mají také svoji ekonomickou stránku. Dle [17] jsou procentuální náklady na jednotlivé fáze života software následující:

Analýza požadavků	3 %
Specifikace	3 %
Design	5 %
Programování	7 %
Testování	15 %
Údržba	67 %

Z výše uvedeného je zřejmé, že na testování produktu je vynaložena nemalá část rozpočtu. Proto je potřebné minimalizovat výdaje pomocí zlepšení kvality a efektivity průběhu testování. Množství chyb, které se v softwarovém produktu nepodaří odhalit ve fázi testování, má navíc vliv na průběh údržby software a tím i na její náklady. Adekvátní a správně prováděné testování šetří rozpočet i jiným způsobem. Kniha [17] cituje výzkum, který hledal souvislost mezi náklady na opravení chyby a včasností jejího odhalení. Z výzkumu vyplývá, že cena za opravu nalezené chyby stoupá v průběhu života software exponenciálně.

2.2 Dělení testování

Testování můžeme dělit do několika skupin - podle fází, ve kterých se testování provádí, podle způsobu testování či podle komplexity testů. Dle [17] patří mezi základní dělení rozdělení na *glass box testování* (někdy též *white box testování*) a *black box testování*. Podle velikosti částí testovaného kódu můžeme dělit testování na *inkrementální testování* a *big bang testování*. Dalším kritériem pro dělení testů je přístup k programu a jeho zdrojovým kódům. Můžeme rozlišovat *statickou analýzu* a *dynamickou analýzu*.

2.2.1 Glass box testování

Glass box testování, někdy také nazývané *testování podsystému* (anglicky *subsystem testing*), je takový druh testování, kdy člověk navrhující testy aktivně využívá své znalosti zdrojových kódů pro vytváření testů. Tento přístup může zajistit téměř úplné pokrytí průchodů zdrojovými kódy programu, a proto je možné otestovat většinu stavů, do kterých se program může dostat [16]. Glass box testování však může provádět pouze člověk, který zdrojovým kódům testovaného programu rozumí, nejlépe jejich tvůrce. Testy jsou navrhovány tak, aby zkoumaly vnitřní elementy a použité struktury [11]. Protože glass box testování může být velice časově náročné, jsou nejčastěji testovány malé části programu - moduly, funkce a tak podobně [11]. Glass box testování se zaměřuje na vnitřní strukturu testovaného kódu. Testy jsou často cíleny na testování konkrétních větvení, výrazů.

Důležitým kritériem u glass box testování je úroveň *pokrytí* (anglicky *coverage*). Pokrytí udává procentuální velikost otestované části daných prvků programu. Existuje několik různých kritérií pokrytí, přičemž glass box testování se může soustředit na kterékoliv z nich. Můžeme rozlišovat například pokrytí řádků kódu, pokrytí větvení, pokrytí vstupní domény, pokrytí funkcí, pokrytí hodnot argumentů a další. Cílem glass box testování je vytvořit testovací sadu s daným pokrytím.

Znalosti potřebné pro vytváření testovací sady pomocí principů glass box testování jsou často dostupné až během návrhu konkrétní implementace programu, čili později, než jsou dostupné informace potřebné pro testovací sadu založenou na black box testování. Proto fáze glass box testování začíná později [11].

2.2.2 Black box testování

Black box testování je takový způsob testování, kdy člověk navrhující testy zkoumá reakci programu na různé vstupy. Návrhář black box testů nezná (nepotřebuje znát) podobu zdrojových kódů, protože zkoumá výsledek jako celek [11]. V tomto přístupu k návrhu testů není cílem pokrýt všechny možné průchody zdrojovými kódy programu.

Při tomto druhu testování není potřeba znát strukturu mechanismů programu - důležitá je pouze znalost, jak se má program chovat a jak má reagovat. Testy mohou být specifi-

kovány pomocí *vstupně-výstupních diagramů* či přesně definovanou množinou podmínek, které musí platit před spuštěním testu, a podmínek, které musí platit po spuštění. Tento způsob testování je užitečný pro odhalování chyb specifikace [11]. Používá se nejčastěji pro integrační a akceptační testování. Někdy je také možné black box testování spojit s analýzou chování testovaného programu. Protože může být časově náročné (někdy nemožné) vyzkoušet reakci programu na veškerý možný vstup, je potřeba navrhnout testovací sadu tak, aby bylo nalezeno pokud možno co nejvíce chyb programu.

Důležitou vlastností při návrhu black box testů je proto množina vstupních hodnot. Každý program má definiční obor, ze kterého se tyto hodnoty primárně vybírají. Vstupní hodnoty by měly pokrýt co nejvíce případů, zároveň by jich mělo být co nejméně. Ilene Burnstein v knize Practical Software Testing [11] proto navrhuje několik možností tohoto výběru. Při *náhodném testování* návrhář náhodně vybírá hodnoty z definičního oboru. Jedná se o nejméně spolehlivou metodu. Vybraný vzorek totiž nemusí být dostačující pro úplné otestování software. Dalším problémem této metody je fakt, že hodnoty jsou vybírány pouze z definičního oboru. Z tohoto důvodu není otestována reakce programu na nevalidní vstup. Při *výběru hodnot založeném na rozkladu na množině* je definiční obor rozdělen na konečný počet množin hodnot, pro které je vždy chování programu stejné. Z každé této skupiny je do množiny vstupů programu vybrán reprezentativní prvek. Výstup programu pro tento prvek je stejný jako výstup programu pro všechny prvky skupiny, ze které pochází. Můžeme proto předpokládat, že selhání testu pro tuto hodnotu znamená selhání testu pro celou skupinu. Pokud se při testování tohoto prvku chyba nenalezne, nebyla by nalezena vstupem žádného dalšího prvku z dané skupiny. Pravidlem je, že návrhář testů musí vzít v úvahu validní i nevalidní vstup. Tento přístup má několik výhod [11]:

- významně se redukuje počet spouštěných testů,
- návrhář testů je veden pro výběr takové množiny hodnot, aby nalezl co nejvíce chyb a
- umožní pokrytí definičního oboru menší množinou hodnot.

Dalším principem výběru testovacích hodnot je *analýza krajních hodnot*. Tato metoda může být použita zároveň s výběrem hodnot založeném na rozkladu na množině. Rozšiřuje tuto metodu takovým způsobem, že vybraný reprezentativní vzorek ze skupiny jsou hodnoty blízké hranicím této skupiny. Existuje množství jiných principů pro výběr vstupů pro black box testování. Mezi nejznámější patří *metoda hádání chyb*, kdy programátor využívá svých zkušeností s testováním podobného software pro výběr hodnot, na které bude s největší pravděpodobností testovaný program reagovat chybně.

2.2.3 Inkrementální testování

Program je testován postupně. Nejdříve se testují malé části (jednotkové testování), ty jsou poté testovány dohromady, po větších celcích (integrační testování). Inkrementální testování zajistí snazší nalezení chyby v kódu. Na druhou stranu je však nutné naprogramovat *ovladače*, což je kód, který se bude starat o spouštění těchto částí se správným vstupem, bude kontrolovat výstup, simulovat volání jiných částí z testované části a podobně.

Jednotkové testování

Termín *jednotka* (česky také někdy překládáno jako *komponenta*, anglicky *unit*) vyjadřuje nejmenší možnou práci, která může být přiřazena jednomu vývojáři a může

být plánována a sledována [24]. V procedurálních jazycích se například může jednat o funkci. V objektově orientovaných jazycích se jedná o třídu. Jednotkové testování má výhodu jednoduchosti návrhu a implementace testů a také rychlost ve spouštění, získávání a analýze výsledků těchto testů.

Dle [11] by měly být testy jednotek vyvíjeny za použití glass box i black box testování. Jednotkové testování by mělo být prováděno nezávislou osobou, ne vývojářem této jednotky. Výsledky jednotkových testů by měly být zaprotokolovány a veřejně dostupné. [11] dále uvádí, že jsou jednotkové testy často vytvářeny a spouštěny vývojářem této jednotky a jejich výsledek není veřejně ukládán. Jedná se o špatnou praktiku, které je potřeba se vyhnout [11].

Integrační testování

Integrační testování je takový druh testování, kde jsou komponenty software spojeny do jednoho spolupracujícího celku a právě spolupráce těchto komponent („integrace“ komponent do zbytku systému) je předmětem testování [11]. Integrační testování by mělo navazovat na jednotkové testování. Jakmile je komponenta otestována sama o sobě, je potřeba začít s testováním rozhraní, komunikací komponent. Při integračním testování jsou do testovaného systému přidávány postupně další a další komponenty. Jakmile je integrován kompletní systém, začíná fáze systémového testování.

Nástroje na integrační testování mohou často být součástí *systémů pro průběžnou integraci* (anglicky *continuous integration systems*). Takovýto systém monitoruje repozitáře se zdrojovými kódy a při každé změně provede automatické spuštění testů, kompilování nových verzí programu atp. Vývojář tak chvíli po uložení zdrojových kódů na server získá zpětnou vazbu o integraci naprogramované komponenty ve formě výsledků testů a také nejnovější verzi programu.

Systémové testování

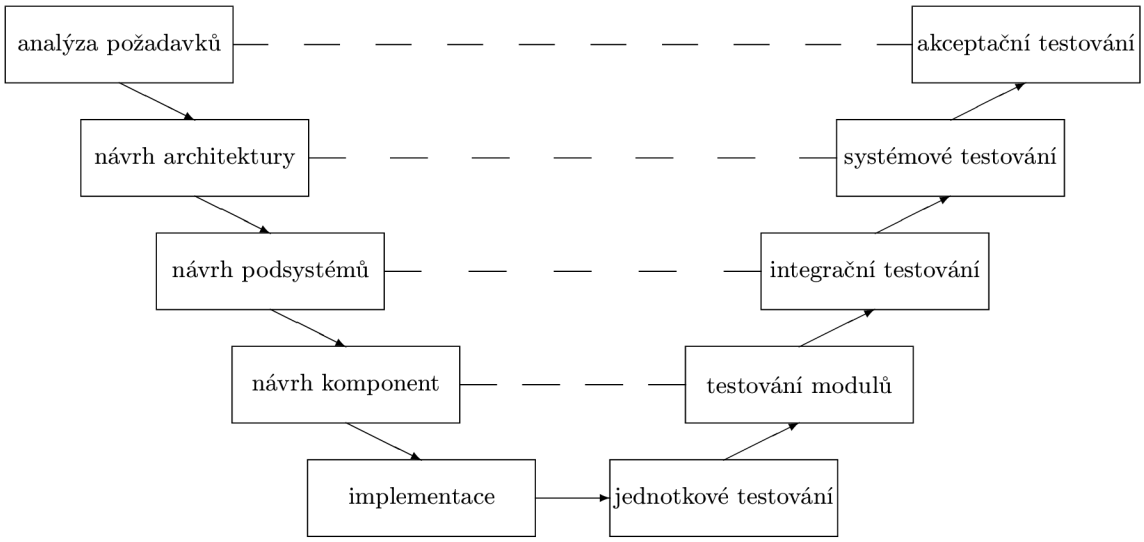
Systémové testování je testování výsledného produktu, systému, dohromady. Jedná se o logické pokračování integračního testování. Systémové testování je testování validační [17].

2.2.4 Statická analýza

Statická analýza je takový způsob analýzy, kdy testovaný program není přímo spouštěn, je pouze zkoumán jeho zdrojový kód [17]. Je prováděna mnoha nástroji, nejčastěji překladačem a linkerem při překladu programu. Může být také prováděna lidmi - čtením zdrojového kódu. Některé metodiky kladou na statickou analýzu lidmi velký důraz, příkladem je párové programování, použité v metodice Extrémní programování.

2.3 Principy V-modelu při testování

Při použití V-modelu pro testování je z každé fáze vývoje software vyvozen odpovídající stupeň testování [8]. Na začátku vývoje je analýza požadavků zákazníka. Souběžně s ní by měly vznikat *akceptační testy*, které budou spouštěny před předáním software zákazníkovi. Následuje návrh architektury programu a zároveň s ní také testy pro *systémové testování*, které bude vyústění integračního testování. Po architektonickém návrhu přichází konkrétnější návrh rozdělení do jednotlivých komponent a také *integračních testů* pro testování správnosti komunikace těchto komponent. Po něm přichází design komponent a návrh



Obrázek 2.1: V-model

s ním spojených *testů modulů*. Nakonec přijde na řadu samotná implementace a testy pro *jednotkové testování*. V testovací fázi vývojového modelu jsou testy prováděny v opačném pořadí, než byly vytvářeny. Jednotkové testování začíná zároveň s implementací, následuje testování modulů, integrační testování, systémové testování a nakonec akceptační testování. Příklad V-modelu je na obrázku 2.1.

2.4 Automatizace testování

Automatizace testování patří mezi možnosti, jak snížit cenu a dobu potřebnou pro testování software. Určitá forma automatizace je vhodná, pokud testování produktu zahrnuje stejné, často opakované úkony [14]. Velmi užitečná je automatizace regresního[1] a akceptačního testování [17]. Dle knihy Automated Software Testing [14] je užitečné využití *testovacího frameworku*. Testovací framework je infrastruktura funkcí a nástrojů pro snazší vývoj, spouštění a analýzu testů. Jeho použití umožní soustředit se pouze na samotný vývoj testů.

Důležitým úkolem je získávání testovacích případů. Zahrnuje nejčastěji ruční vytvoření testovacích případů a jejich následné automatické (a často i pravidelné) spouštění. Při vytváření testů pro automatické spouštění může být použito i principů náhodného testování [17], avšak toto testování by nemělo být náhradou testování krajních hodnot, ale jeho doplňkem. Samostatnou kapitolou je automatické vytváření testovacích případů. Takové metody se dotýkají netriviálních témat z oblasti formální verifikace [26] a tato práce se jimi nebude dále zabývat. Automatizovat lze také glass box testování. Existují například nástroje pro automatické generování testovacích případů s co největším pokrytím kódu [23].

[1] testování, zda ve funkčním programu nevznikla chyba jeho úpravami

2.5 Testování uživatelského rozhraní

S rozmachem programů poskytující grafické uživatelské rozhraní se rozmohla nutnost tato rozhraní testovat. S každým aktivním prvkem GUI ale narůstá počet možností průchodu programem, ze kterých se vybírají testovací případy. Ovládání programu pomocí GUI má také svá specifika. Výsledku je dosaženo následováním určitého počtů konkrétních kroků, při tom je důležité zachovat pořadí těchto kroků. Navíc je často tyto kroky nutné provádět i když se program nachází v naprosto odlišných počátečních stavech. Proto testování GUI vyžaduje specializované metody a postupy. Důležitost testování uživatelského rozhraní dokazuje například jeho časté použití při vývoji mobilních aplikací [3, 7].

Zatímco úseky zdrojových kódů mají pevně dané, programem jednoduše použitelné rozhraní, uživatelská rozhraní (dále UI) musí být v první řadě srozumitelná a dostupná koncovému uživateli programu, což má za následek větší obtížnost jejich automatizovaného testování. Při testování GUI vyvstává několik různých problémů. Nástroj pro automatizované testování GUI musí řešit tyto dva body:

1. ovládání UI,
2. reprezentace testovacích případů.

První důležitou vlastností je způsob, kterým testovací nástroj ovládá UI testovaného programu. Odvíjí se od něj nejen obtížnost psaní testů a s ním spojená přehlednost, ale také stabilita testů při změnách uživatelského rozhraní. Nástroj pro automatizované testování UI může pro ovládání používat tyto dvě techniky:

Ovládání GUI s použitím obrazovky

Testující program zasílá UI příkazy pro ovládání tak, jako kdyby jej ovládal opravdový uživatel. Využívá příkazů pro posun myši na určité souřadnice, stisknutí tlačítek myši a psaní na klávesnici [19]. Použití této techniky může být zjednodušeno pomocí předem připravených snímků obrazovky. Testující program porovnává aktuální stav obrazovky se stavem na snímku a na základě toho zasílá příkazy pro ovládání UI [28]. Vyhledávání objektů na snímku obrazovky může být dále vylepšeno použitím knihoven pro počítačové vidění. To umožní porovnáním dvou snímků získat jejich přibližnou procentuální míru shody a vyhledávat části obrazu ve větším obrazu metodou vyhledávání vzorů (anglicky *template matching*) a zajistit tak větší stabilitu testů, které jsou pak méně náchylné na drobné změny testovaného systému [13].

Mezi nevýhody použití snímků obrazovky patří velká závislost na stylu zobrazení. Testy je nutné vytvářet znovu při každé změně barev, písma či grafického stylu aplikace, ač logika ovládání mohla zůstat zachována. Jejich výhodou však je možnost odhalení chyb vykreslování GUI a jeho prezentace uživateli.

Ovládání GUI bez použití obrazovky

Další technika vyžaduje bližší porozumění struktuře zobrazeného UI. Nejrůznějšími metodami (asistivní technologie, ovladače UI) poskytuje vysokoúrovňové funkce pro jeho ovládání, například „otevři nové okno“ nebo „stiskni tlačítko Cancel“. Logickým důsledkem těchto technik je však vyšší závislost na použitém software (zobrazovacím serveru, grafických knihovnách apod.).

Přínos této metody je v její efektivitě, protože není nutné emulovat ovládání uživatelem a odpadá také potřeba použití náročné metody vyhledávání vzorů. Ovládání

na vyšších úrovních však s sebou přináší určité nároky, například nárok na znalost vnitřní struktury GUI, nebo nutnost řešit spolupráci s grafickou knihovou. Mezi její nevýhody také patří nemožnost testovat chyby ve vykreslování grafických prvků.

Dalším bodem, na který se zaměříme, je způsob vytváření a reprezentace testovacích případů. Nezávisle na použité technice ovládání může nástroj používat tyto metody:

Metoda zachycení a přehrání [2]
Jedná se o nejjednodušší a zároveň nejčastěji používanou metodu. V kódu testu jsou uloženy kroky, kterými má program projít jeden za druhým. Častá metoda vytváření těchto testů je použití programu, který zachytává události vývojáře testů a následně je schopen je ve stejném sledu zopakovat [22]. Jedná se o koncept, který je známý ze spousty různých programů jako tzv. *makra*.

Systém plánování a cílů s použitím algoritmů umělé inteligence
Ve článku [21] byla představena metoda použití umělé inteligence pro vytváření testovacích případů. Každý test pak sestává z počátečního stavu, množiny koncových stavů a množiny možných operátorů. Testující program použije metody prohledávání stavového prostoru pro vytvoření testovacích případů, které vedou z počátečního do cílového stavu.

Genetické algoritmy
Jedním z dalších postupů je použití genetických algoritmů. Dle *Toward Automatic Generation of Novice User Test Scripts* [18] je jedním z přehlížených stylů testování takzvané „začátečníkovo testování“. Je zde nastíněna myšlenka, že uživatel-začátečník se uživatelským rozhraním pohybuje nepředvídatelným, ale zato rozumným způsobem. S každým použitím se začátečník stane více poučenější o principu fungování programu a proto se jeho používání stává více a více optimálním. Genetické algoritmy jsou vhodným prostředkem simulace takového chování.

[2] v anglické terminologii známá jako „record & replay“

Kapitola 3

Dostupné nástroje pro testování GUI

Nástroj pro automatizované testování GUI se může skládat z více částí. Nejdůležitější součástí je kód, který s grafickým rozhraním přímo pracuje. Je důležité, aby tato část měla dobře použitelné API[1]. Toho je většinou dosaženo dvojím způsobem - daný nástroj má buď přímo vestavěný programovací jazyk, který má dostupné požadované funkce, a nebo je tento nástroj knihovnou (modulem) některého existujícího programovacího jazyka. Častým rysem nástrojů pro ovládání GUI s otevřeným zdrojovým kódem bývá použití některého dynamického programovacího jazyka (Python, Perl atp.) [4, 1].

U nástrojů pro interakci s GUI můžeme rozlišit způsob, jakým testovací program GUI ovládá. Nejčastější způsob je simulace událostí klávesnice a myši. Například otevření dialogu pro uložení souboru se poté skládá z několika úkonů (najetí kurzorem na danou pozici, stisknutí tlačítka myši, najetí kurzorem na jinou pozici, opětovné stisknutí tlačítka myši). Jiný způsob je přímý přístup ke grafickým prvkům pomocí vnitřních rozhraní grafických knihoven či pomocí meziprocesové komunikace (např. použití D-Bus).

Další požadovanou částí těchto nástrojů je testovací framework. K mnoha programovacím jazykům se váží de facto standardní testovací knihovny. Struktura a vlastnosti těchto knihoven často vychází z návrhu představeného Kentem Beckem v jeho knize *Test-Driven Development: By Example* (česky *Programování řízené testy* [10]). Tyto knihovny se souhrnně označují *xUnit*. Jejich výhodou je jejich rozšířenost, která zároveň znamená vysokou dostupnost nejrůznějších nástrojů, které s nimi mohou pracovat.

3.1 Nároky na použitý nástroj

V rámci této práce bylo prozkoumáno několik nástrojů pro testování GUI. Jelikož je testování instalátoru operačního systému činnost náročná na použité postupy, bylo nutné definovat několik kritérií, které by měl použitý nástroj podporovat. Jedná se o tato kritéria:

1. rozdělení hostitelského a hostovaného systému,
2. co nejmenší nároky na testovaný systém,
3. použití vhodného způsobu rozpoznávání,

[1] Application Programming Interface, rozhraní pro programování aplikace

4. použití virtualizace.

Rozdělení na hostitelský a hostovaný systém
Toto kritérium vyjadřuje požadavek na zajištění takové funkcionality, aby použitý nástroj mohl být spuštěn na jiném stroji, než který bude tímto nástrojem testován. Tento návrh přináší některé výhody. Testovaný systém bude testováním minimálně ovlivněn a dále bude možné získat informace i o takovém systému, v němž dojde v průběhu testování k fatální chybě. Nevýhodou tohoto přístupu mohou být zvýšené nároky na hardwarové vybavení.

Co nejmenší nároky na testovaný systém
Není žádoucí, aby se testovaný systém musel pro potřeby testování jakkoliv upravovat. Tyto úpravy (jako například doinstalace dodatečného software) mohou způsobit odlišné chování systému při testování. Jedním z kritérií je proto kritérium na co nejmenší nároky na testovaný systém.

Tento požadavek má také další důvod. Je potřeba si uvědomit, že při testování instalace operačního systému jsou testovány operační systémy dva. Instalace operačního systému Linux totiž probíhá nejčastěji z tzv. *live OS*. To je takový systém, který je spuštěný z přenosného média a načtený do paměti RAM, aniž by musel zasahovat do systému uloženého na pevném disku. V tomto prostředí je pak spuštěn instalační program. Po nainstalování se počítač přepne do nového operačního systému, ve kterém dokončí instalaci, vytvoří uživatele atp. Proto by bylo užitečné, aby vybraný nástroj dokázal ovládat oba operační systémy, nejlépe bez jakékoliv potřebné změny v kódu testů.

Použití vhodného způsobu rozpoznávání
Při testování metodou zachycení a přehrání je důležitým aspektem způsob, jakým testovací nástroj nachází očekávaný snímek v aktuálním snímku systému. Některé nástroje obsahují pouze jednoduché bitové porovnávání, zatímco jiné používají sofistikovanějších nástrojů, jako například knihovny pro počítačové vidění (CV, computer vision). Metoda bitového rozdílu může mít katastrofální následky - vytvořené testy je nutno předělávat při změně byť jediného pixelu. Funkce knihovny pro počítačové vidění zajistí fungování vytvořených testů i při lehce odlišných podmínkách. Tato funkčnost je pro použitý nástroj klíčová. Pokud bude testování probíhat na dvou různých systémech (hostitelský a hostovaný, viz první kritérium), bude díky počítačovému vidění možné přenášené snímky ztrátově komprimovat. Nástroje implementující bitové porovnávání by při tomto způsobu použití mohli lehce selhat. Mezi nevýhody počítačového vidění při automatizovaném testování GUI patří možnost *falešně pozitivních* výsledků.

Použití virtualizace
Toto kritérium je spíše logickým vyústěním předešlých kritérií. Pokud bude testování probíhat na virtualizovaném počítači, testování neovlivní testovaný systém, testovat bude možno systém v neupravené podobě.

Posledním aspektem je požadavek, vyplývající přímo ze zadání. Použitý nástroj musí být co nejsnadněji spustitelný na operačním systému Linux. V ideálním případě by mělo být možné vytvořit instalační balíček tohoto programu pro distribuci Fedora.

3.2 Příklady dostupných nástrojů

S ohledem na výše uvedená fakta byly prozkoumány a zhodnoceny tyto nástroje:

os-autoinst

URL: http://os-autoinst.org/
Licence: GNU/GPL v2.0

Program os-autoinst je komunitní projekt. Mezi jeho nejvýznamnější uživatele patří projekt openSUSE[2]. Nástroj je psaný v jazyce Perl a v témže jazyce se také píší samotné testy. Při psaní testů se nepoužívá žádný standardizovaný testovací framework a navíc formát výstupu testů není pořádně dokumentován. Os-autoinst používá virtualizační nástroje (jmenovitě VirtualBox od firmy Oracle či QEMU) pro testování na virtualizovaném systému. Nevýhodou může být, že pro interakci s virtualizační technologií nepoužívá žádné unifikované rozhraní, jako například knihovnu *libvirt*[3]. Na obrázku 3.1 je vidět webové rozhraní *OpenQA*, které slouží pro zobrazování výsledků testů z programu os-autoinst. Příklad testu psaného pro program os-autoinst je v příloze A.1.

Problém nastává při bližším prozkoumání způsobu, který program používá pro porovnávání snímků obrazovky. Program os-autoinst původně nepoužíval žádný způsob počítačového vidění pro rozpoznávání snímků a metodu vyhledávání vzorů. Očekávané snímky byly reprezentovány pouze výstupem z md5sum hešovací funkce. V posledních několika měsících se projekt snaží začlenit použití knihovny OpenCV[4], ale tato možnost je pouze v experimentální fázi, stále neexistuje žádná dokumentace k použití těchto funkcí.

Autopilot

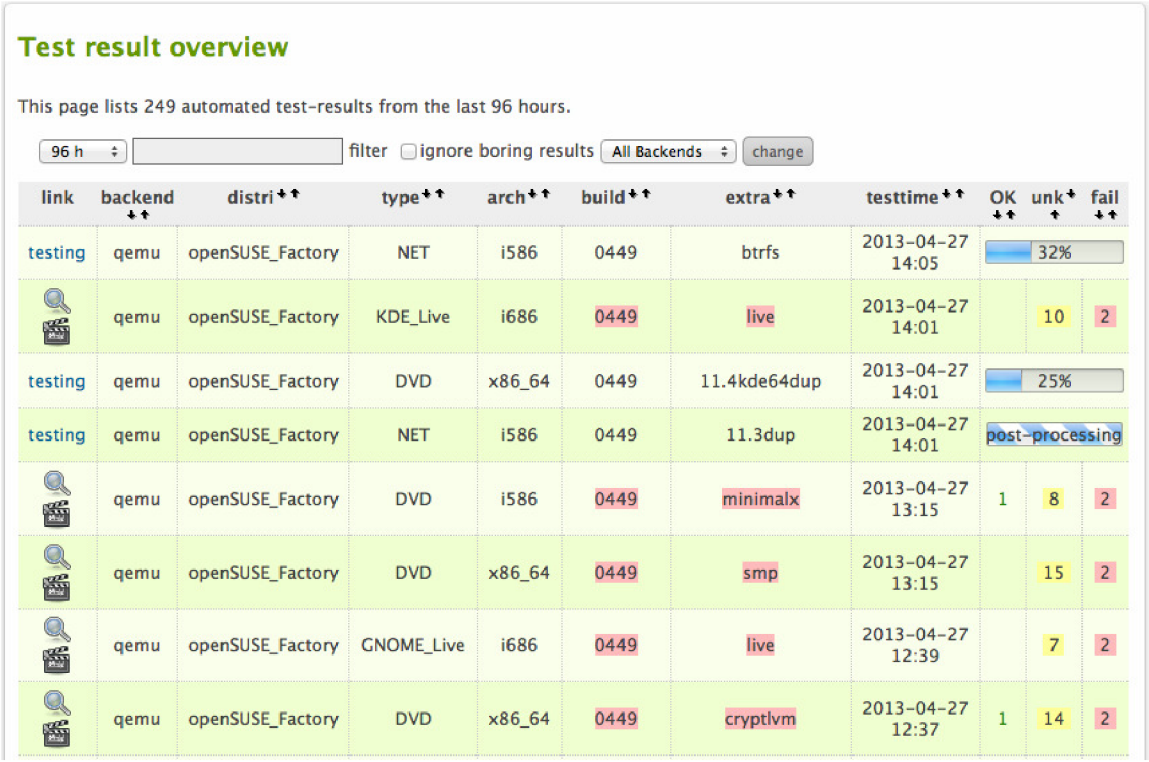
URL: https://wiki.ubuntu.com/Unity/QA/Autopilot
Licence: GNU/GPL v3.0

Autopilot je produkt firmy Canonical, která stojí za vývojem linuxové distribuce Ubuntu. Je součástí grafického uživatelského prostředí Unity, k jehož testování také primárně slouží. Program je implementován v jazyce Python. Jako testovací framework používá standardní knihovnu Pythonu jménem *unittest*.

Testovací skript, psaný pro program Autopilot, se skládá ze dvou částí. První část se stará o interakci s uživatelským rozhraním Unity. Tato část se nazývá emulátor. Grafické prvky prostředí Unity mají své emulátory, které poskytují rozhraní pro jejich introspekci a ovládání. K tomuto účelu používá framework pro meziprocesovou komunikaci *D-Bus*. D-Bus umožňuje ovládat uživatelské rozhraní na vyšší úrovni, než pohybem myši. Poskytuje příkazy pro přímé ovládání konkrétních grafických prvků, například funkci pro otevření souboru pomocí dialogu atp. Druhou částí jsou samotné testy, které jsou psány v testovacím frameworku unittest. Informace o stavu prostředí se získávají pomocí programu D-Bus. Funkce `get_state`, poskytovaná API programu Autopilot, vrací *strom introspekce*. V tomto stromu je uložen stav všech komponent grafického prvku, na jehož strom introspekce jsme se dotázali. Autopilot poskytuje jednoduchou syntaxi na dotazování, podobnou jazyku *XPath*.

Jeho nevýhodou je přílišná vazba na prostředí Unity a také přílišné svázání s operačním systémem. Ze své podstaty nemůže být spuštěn na jiném systému než je ten, který

[2] http://openqa.opensuse.org/
[3] http://libvirt.org/
[4] https://github.com/bmwiedemann/os-autoinst/commit/9ceed2a8d8fd0



Test result overview

This page lists 249 automated test-results from the last 96 hours.

96 h | filter | ignore boring results | All Backends | change

link	backend	distri	type	arch	build	extra	testtime	OK	unk	fail
testing	qemu	openSUSE_Factory	NET	i586	0449	btrfs	2013-04-27 14:05	32%		
	qemu	openSUSE_Factory	KDE_Live	i686	0449	live	2013-04-27 14:01		10	2
testing	qemu	openSUSE_Factory	DVD	x86_64	0449	11.4kde64dup	2013-04-27 14:01	25%		
testing	qemu	openSUSE_Factory	NET	i586	0449	11.3dup	2013-04-27 14:01	post-processing		
	qemu	openSUSE_Factory	DVD	i586	0449	minimalx	2013-04-27 13:15	1	8	2
	qemu	openSUSE_Factory	DVD	x86_64	0449	smp	2013-04-27 13:15		15	2
	qemu	openSUSE_Factory	GNOME_Live	i686	0449	live	2013-04-27 12:39		7	2
	qemu	openSUSE_Factory	DVD	x86_64	0449	cryptlvm	2013-04-27 12:37	1	14	2

Obrázek 3.1: OpenQA, webové rozhraní pro os-autoinst

testuje. Ze stejného důvodu také pro testování nepoužívá virtualizovaný systém. Ukázka použití programu Autopilot v testu, který testuje prvek prostředí Unity zvaný *Home Lens*, je v příloze A.2.

Sikuli

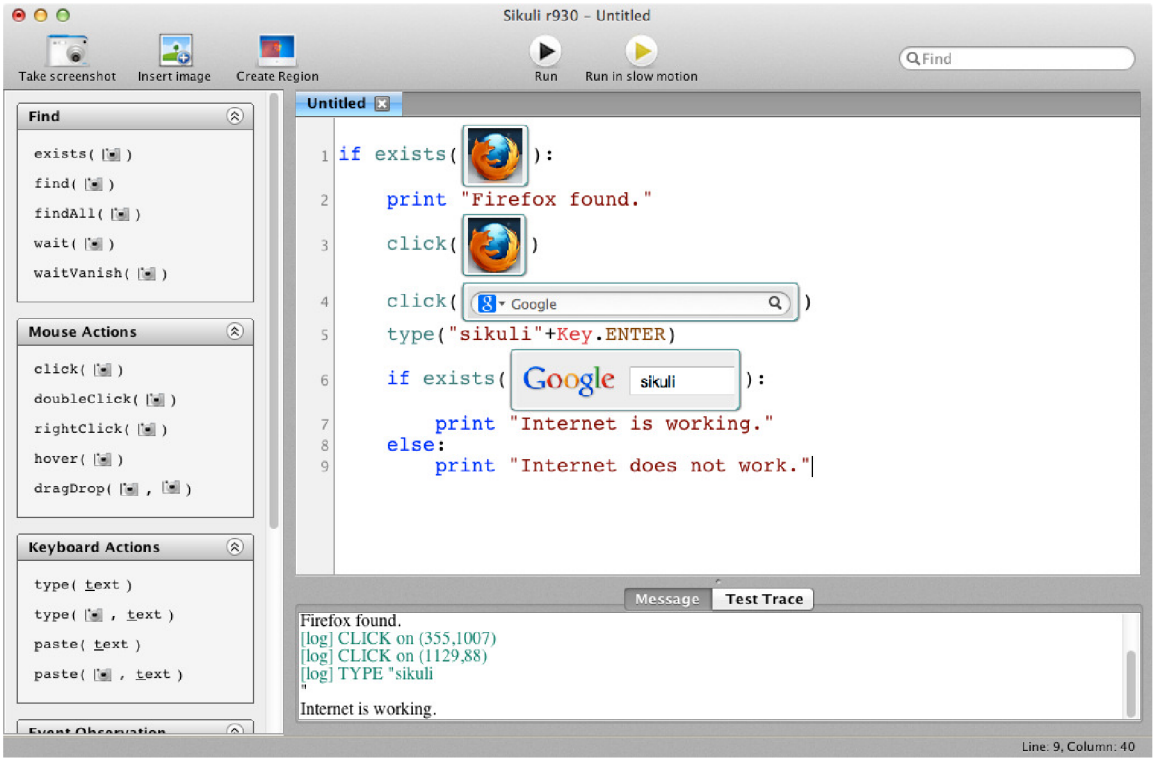
URL: http://www.sikuli.org/
Licence: MIT

Nástroj, vyvinutý na univerzitě MIT, sloužící primárně k automatizaci ovládání GUI, ačkoliv může být použit i k jeho testování. Sikuli je známým a široce používaným programem. Je napsán v Javě, vestavěný jazyk pro psaní testů je *Jython*[5]. Obsahuje i vestavěné vývojové prostředí, které je vidět na obrázku 3.2. Výhodou je také fakt, že se jedná o multiplatformní software. Obsahuje knihovny na ovládání tří operačních systémů, Microsoft Windows, Linuxu a OS X. Ač poskytuje příjemné UI pro vytváření testů, může fungovat i jako konzolový program. Program Sikuli používá knihovnu *JUnit*, což je testovací framework pro programovací jazyk Java. Zajímavou vlastností Sikuli je možnost definovat reakce na různé události, například klávesovou zkratku, zobrazení grafického prvku na obrazovce a podobně. Sikuli používá knihovnu OpenCV pro vyhledávání vzorů a různé platformově-specifické knihovny k získávání snímků obrazovky.

Problémem však může být fakt, že Sikuli jako takové není stavěno na rozdělení systémů hosta a hostitele. Ačkoliv vznikaly snahy, které měly za cíl spojit Sikuli a protokol VNC, vzniklý fork *sikuli-vnc* je v dnešní době dále nevyvíjený [12]. Dalším problémem je závislost

[5] implementace Pythonu v Javě

na velmi staré verzi knihovny OpenCV, kterou je problém zkompilovat na novějším systému. Jedná se o známou a nahlášenou chybu[6]. Součástí příští verze programu Sikuli má být závislost na nové verzi OpenCV, nicméně její vydání se stále odkládá.



Obrázek 3.2: Vývojové prostředí programu Sikuli v OS X

Xpresser

URL: https://wiki.ubuntu.com/Xpresser
Licence: GNU/LGPL

Program Xpresser je produkt, jehož vývoj je zaštiťován firmou Canonical. Jedná se o modul pro programovací jazyk Python, který umožňuje automatizované ovládání GUI. Pro porovnávání a rozpoznávání snímků na obrazovce používá *SimpleCV*, což je knihovna postavená nad knihovnou OpenCV. Pro ovládání grafických prvků používá příkazy pro simulaci používání myši a klávesnice z knihoven *cairo* a *GDK*. Využívání těchto knihoven znemožňuje užití knihovny Xpresser pro vzdálené ovládání. K SimpleCV se váže nahlášená chyba[7], která brání použití Xpresseru na starších verzích distribuce Fedora.

Výhodou je fakt, že Xpresser je rozsahově malá (přibližně 600 řádků kódu) a nenáročná knihovna. Xpresser je proto adeptem pro naše účely, jelikož by byly úpravy vedoucí k požadované funkcionalitě snadno začlenitelné do stávajícího kódu. Bezproblémová je také integrace s testovacím frameworkem unittest. V příloze A.4 je ukázka použití této knihovny.

[6] https://bugs.launchpad.net/sikuli/+bug/949808
[7] https://bugs.launchpad.net/xpresser/+bug/1060120

Dogtail

URL: https://fedorahosted.org/dogtail/
Licence: GNU/GPL v2.0

Dogtail je komunitní projekt, jeho vývoj je řízen firmou Red Hat. Je psán v programovacím jazyce Python a je primárně určen jako testovací framework pro automatizované testování GUI. Používá tzv. *asistivní technologie*. Asistivní technologie jsou takové technologie, které usnadňují použití programů osobám s fyzickým handicapem. Součástí grafických knihoven Qt a GTK+ je *Assistive Technology Service Provider Interface* (AST), rozhraní pro použití programů s asistivními technologiemi (jako je například čtečka obrazovky apod.). Toto rozhraní používá program Dogtail pro obsluhu grafických prvků. Příklad testu psaného za pomocí programu Dogtail je v příloze A.3.

Dogtail obsahuje užitečné nástroje na nahrávání skriptů a prohlížení hierarchie grafických prvků. Nevýhodou použití asistivních technologií je však přílišná vazba na systém. Dogtail může ovládat pouze programy psané s knihovnami Qt nebo GTK+. Dogtail navíc nedokáže používat ovládací prvky, pokud nemají definovány informace pro AST. Program Anaconda je sice psán za pomocí grafické knihovny GTK+, ale jeho grafické prvky jsou vytvořeny na míru programu a neobsahují AST informace, což znamená, že se program Dogtail pro ovládání programu Anaconda použít nedá. Další překážkou je, že program Dogtail není uzpůsoben pro ovládání jiného systému než je ten, na kterém je spuštěn.

Další dostupné nástroje

Existuje široká škála dalších nástrojů na testování GUI, které však nejsou vhodné pro naši úlohu. Známým a často používaným testovacím frameworkem je *Selenium*[8]. Tento projekt slouží pro automatizované testování grafického rozhraní webových aplikací. Poskytuje možnost rozdělení klient-server, vlastní jazyk pro psaní testů stejně jako možnost psát testy v jiných programovacích jazycích. Obsahuje také rozšíření pro prohlížeč Firefox, které slouží jako vývojové prostředí testů.

Mezi další projekty, které používají asistivní technologie pro ovládání grafických komponent, patří například *Linux Desktop Testing Project*[9]. Na programovací jazyk Java a jeho vývojové prostředí je orientována spousta nástrojů pro testování GUI, například *Maveryx*, *CubicTest* či *Jemmy*.

Poslední možnost, která připadala v úvahu při automatizované instalaci distribuce Fedora je použití tzv. *Kickstartu*[10]. Kickstart je soubor s nastavením instalace distribuce Fedora, který je čten programem Anaconda. Jednou z nevýhod tohoto přístupu je však absence jakékoliv podpory pro testování.

[8] http://docs.seleniumhq.org/
[9] http://ldtp.freedesktop.org/wiki
[10] http://fedoraproject.org/wiki/Anaconda/Kickstart

Kapitola 4

Základní koncept testovací sady programu Anaconda

Vytváření testovacích sad pomocí umělé inteligence či genetického programování vyžaduje hlubší porozumění těmto principům. Výsledkem může být lepší vzorek testovací sady, ale za cenu výrazně těžší implementace. Pro testování programu Anaconda byl proto vybrán ruční návrh testovacích sad. Při návrhu testovací sady musíme brát v potaz metodu použitou pro testování stejně jak charakteristiku testovaného software. Testovací sada by měla být co nejmenší a zároveň by měla pokrýt co nejvíce možností, stavů, do kterého by se mohl testovaný software dostat. Pro pokrytí implementované testovací sady bylo vybráno kritérium pokrytí všech hlavních aktivit programu Anaconda. Nejprve provedeme analýzu testovaného software.

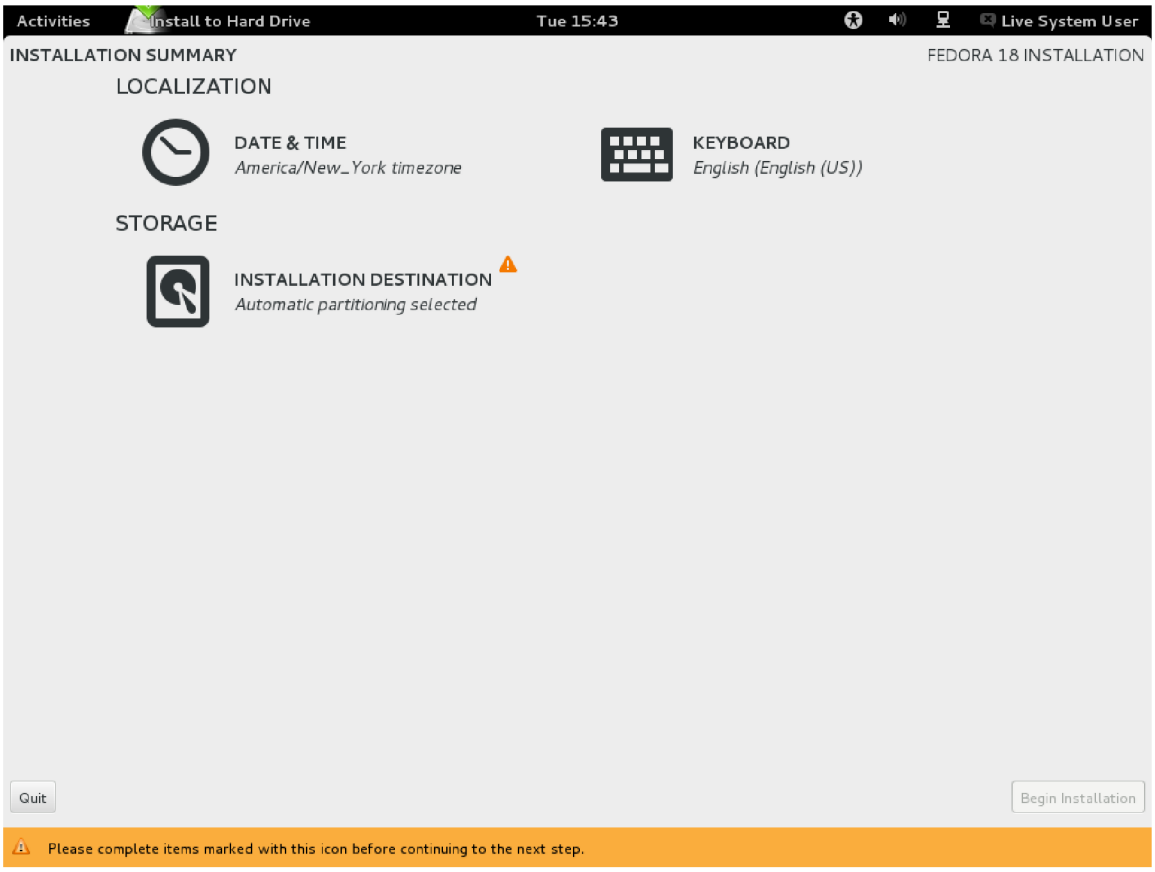
4.1 Program Anaconda

Linuxová distribuce Fedora je instalována z live systému, který je celý načten do paměti RAM. V tomto systému je spuštěn program Anaconda, který slouží pro instalaci systému - kopírování souborů, nastavení nainstalovaného systému, vytvoření administračního účtu atp. Je nezbytné, aby tato kritická část byla dostatečně uživatelsky přívětivá a zároveň stabilní.

Program Anaconda prošel s Fedorou verze 18 změnou filosofie ovládání. Po spuštění instalace je uživateli nabídnut seznam možného nastavení, takzvaný *hub*, ze kterého je možno v libovolném pořadí vstoupit do nastavení vlastností instalovaného systému (klávesnice, čas a datum atp.), takzvaných *spoke*. Rozložení a vzhled hubu Anacondy, která je součástí systému Fedora 18[1] je na obrázku 4.1. Do testovací sady bude potřeba zahrnout testování každé obrazovky nastavení, které jsou zobrazené v hlavní nabídce. Protože jednotlivá nastavení na sobě mohou být závislá, je potřeba otestovat jejich různé kombinace.

Dle zkušeností je důležité testovat opakovaný vstup do stejných obrazovek nastavení. Naopak do testovací sady nebude zahrnuto testování jiného jazyka, než je implicitně vybraná angličtina. Chování programu Anaconda je v různých jazycích stejné, ale velikost testovací sady by při přidání jednoho dalšího jazyka narostla na dvojnásobek.

[1] Pro implementaci testů byla vybrána Fedora verze 18, neboť v době psaní testů nebylo možné Fedoru verze 19 pomocí dostupných obrazů vůbec nainstalovat.



Obrázek 4.1: Hlavní obrazovka (hub) Anacondy z Fedory 18

4.2 Průchod instalací distribuce Fedora

Jednotlivé testovací případy navržené testovací sady provedou instalaci distribuce Fedora od prvního spuštění až po spuštění nainstalovaného systému. Tímto způsobem bude vyzkoušeno fungování jednotlivých voleb stejně jak schopnost nainstalovat při daném nastavení systém a také funkčnost samotného nainstalovaného systému. Fáze, kterými systém projde od prvního spuštění až po načtení operačního systému, jsou následující:

1. Boot (načítání) systému, ze kterého bude spuštěna instalace.
2. Spuštění programu Anaconda.
3. Výběr jazyka instalace a nainstalovaného systému.
4. Zobrazení hlavní obrazovky (hubu) programu Anaconda.

   Zde mohou být změněny a otestovány následující kroky:

   - Nastavení data, času a časové zóny.
   - Nastavení rozložení klávesnice instalovaného systému.
5. Nastavení rozložení disku. Rozložení disku může být buď navržené automaticky, nebo zvolené ručně. Lze zde také nastavit šifrování disku.

6. Spuštění a průběh instalace. V rámci instalace je zvoleno heslo uživatele *root*.

7. Restartování počítače po dokončení instalace a načtení programu *firstboot*.

8. Program firstboot obsahuje průvodce vytvořením nového uživatele. V rámci principu analýzy krajních hodnot lze otestovat vytvoření uživatele s diakritikou ve jméně. Tento případ byl také doporučen při Fedora Gnome Test Day[2].

9. První přihlášení nově vytvořeným uživatelem.

V každé fázi může být více různých voleb či nastavení, které je nutno otestovat. Některé fáze, například nastavení data a času nebo nastavení rozložení klávesnice, mohou být přeskočeny. Vytvořená testovací sada by měla pokrývat co nejvíce různých kombinací průchodů těmito fázemi.

4.3 Kontrola po instalaci

Pro otestování správného fungování Anacondy je důležité zaměřit se také na nastavení, které se před instalací mění. Jelikož se však mění nastavení systému, který teprve bude nainstalován, je jedním z možných řešení požadavek, aby se ke každé změněné volbě vázala také funkce, která v nainstalovaném systému zkontroluje, zda došlo k jejímu správnému nastavení. Proto po fázi prvního přihlášení do nainstalovaného systému musí ještě následovat fáze, ve které se v běžícím systému zkontroluje vše potřebné.

Kontrolu nastavení po instalaci není potřeba provádět u každé volby. U velkého množství nastavení je dostatečným důkazem o jejich fungování prostý fakt, že je nainstalovaný systém schopen se načíst a přihlásit uživatele. K takovému nastavení patří například šifrování disku nebo vytvoření uživatelského účtu.

[2] http://fedoraproject.org/wiki/Test_Day:2013-03-21_Gnome_3.8

Kapitola 5

Implementační detaily automatizované testovací sady

Pro implementaci navržené sady testů byl použit nástroj Xpresser, který však bylo nutno upravit, aby vyhovoval všem potřebným kritériím. Protože potřebných testovacích případů je velmi mnoho kvůli velkému množství možných kombinací nastavení, jsou testovací případy generovány dynamicky. Jako testovací framework je použita knihovna unittest, která má však některá omezení, které bude nutné překonat.

5.1 Charakteristika použitého nástroje

Žádný ze zkoumaných nástrojů neposkytuje plnou požadovanou funkčnost se zachováním snadné integrace s linuxovou distribucí Fedora. Díky otevřeným zdrojovým kódům je však možné jeden z výše uvedených nástrojů podle potřeby upravit. Pro tento účel byl vybrán nástroj Xpresser. Jedná se totiž o projekt s dostatečně strukturovanými a přehlednými zdrojovými kódy, aktivní komunitou vývojářů a dostačující funkčností. Ty části, se kterými byly na distribuci Fedora potíže jsou zároveň těmi částmi, které byly nahrazeny kvůli použití vzdáleného ovládání.

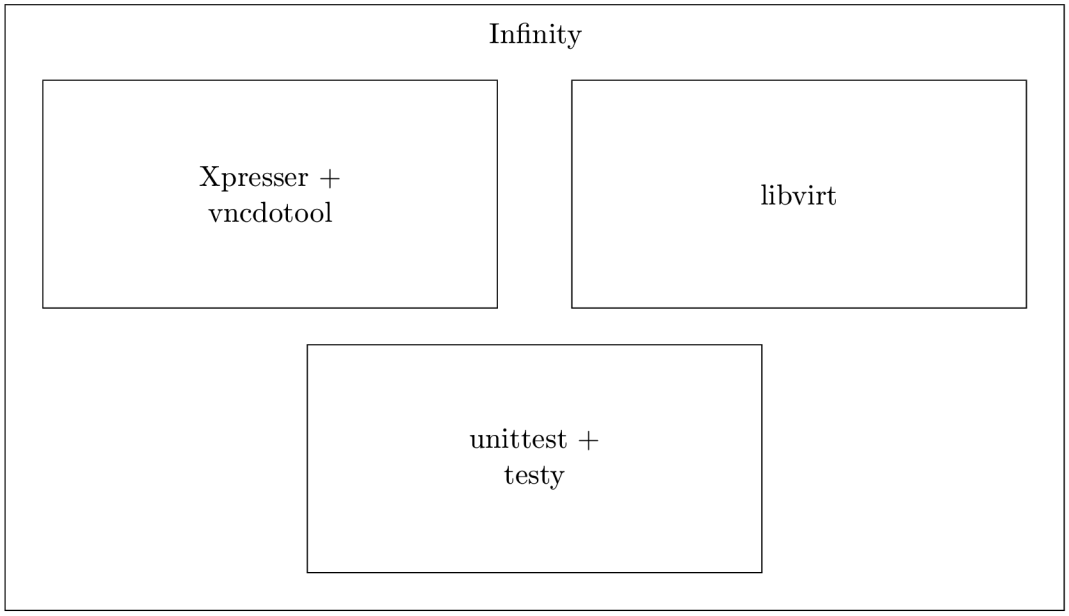
Výsledkem je program v jazyce Python jménem *Infinity*, který používá upravený kód Xpresseru pro ovládání GUI, dále obsahuje funkce pro virtualizaci. Pro účely testování program používá standardní knihovnu Pythonu jménem unittest. Struktura výsledného programu je vidět na obrázku 5.1.

5.1.1 Vzdálené ovládání

Aby bylo možné zajistit rozdělení na hostitelský a hostovaný systém, bylo nutné změnit způsob zasílání příkazů operačnímu systému a také snímání obrazovky systému. Knihovna Xpresser používala původně pro tento účel knihovny cairo a GDK. Volání funkcí těchto knihoven lze na daných místech nahradit voláním, které bude tyto příkazy zasílat jinému systému. Zároveň je potřeba vytvořit kód pro získávání snímků obrazovky cílového systému. Pro tento úkol byly zvažovány dvě možnosti - použití programu typu VNC nebo použití protokolu SPICE.

VNC

VNC je systém pro sdílení pracovní plochy. Používá protokolu *RFB* pro přenos informací o událostech myši a klávesnice ze systému, na kterém běží VNC klient, na



Obrázek 5.1: Struktura výsledného programu

ovládaný systém. Jedná se o často používané standardní řešení vzdáleného ovládání operačního systému (je používáno například v produktu Apple Remote Desktop [9]). Jeho výhodou je jeho rozšíření. Virtualizační technologie QEMU/KVM i VirtualBox jsou schopné zobrazovat grafický výstup virtualizovaného systému při použití protokolu VNC.

SPICE

SPICE je protokol vyvíjený firmou Red Hat, který slouží jako specializovaný protokol pro ovládání virtualizovaného systému. I přes to, že se zdá být vhodnějším kandidátem pro naše účely, musíme brát v potaz jeho menší rozšíření a s ním spojenou absenci knihovny pro jazyk Python, který by pomocí protokolu SPICE dokázal komunikovat. Z virtualizačních nástrojů podporuje protokol SPICE pouze QEMU/KVM.

Pro implementaci vzdáleného ovládání byl použit systém VNC a to za pomocí konzolového programu, psaného v Pythonu, jménem *vncdotool*[1]. Ač VNC nemá takovou funkčnost jako protokol SPICE, na výsledný program nebude klást žádné omezení a výhodou je existence kompletně konzolového nástroje. K tomuto přístupu se však váže problém. Jedná se o nahlášenou chybu[2], kdy čas od času nedojde k zaregistrování požadované události ovládaným systémem. Do této chvíle jsem nedokázal určit, zda se jedná o chybu v programu vncdotool nebo programu QEMU. Na výsledné testování má naštěstí tato chyba minimální vliv. Při vyhodnocování výsledků testů je však nutno s touto vlastností počítat.

[1] https://github.com/sibson/vncdotool
[2] https://github.com/sibson/vncdotool/issues/13

5.1.2 Rozpoznávání snímků

Knihovna Xpresser používá pro rozpoznávání SimpleCV, což je knihovna postavená nad OpenCV. S knihovnou SimpleCV jsou v distribuci Fedora verze 17 spojené určité potíže[3]. Kód, ve kterém knihovna Xpresser používá knihovnu SimpleCV, však může být nahrazen přímo voláním funkcí z knihovny OpenCV.

Pro rozpoznávání snímků se používá metoda vyhledávání vzorů. Vstupem pro tuto techniku jsou dva obrázky, jeden je považován za šablonu, která se poté vyhledává v druhém obrázku. Výstupem OpenCV je matice, ve které jsou zapsány hodnoty od 0 do 1. Tyto hodnoty vyjadřují podobnost obrázků (0 - žádná, 1 - úplná), pokud by šablona byla do obrázku vložena na souřadnice, které daná pozice v matici výsledků zastupuje. Funkce, které poskytuje knihovna SimpleCV, poté výslednou matici zkontrolují a vrátí seznam souřadnic, na kterých podobnost přesáhla požadovanou hranici. Při testování GUI metodou zachycení a přehrání se nejedná o nutnou funkcionalitu. V našem programu stačí informace, zda se v hledaném obrázku nachází alespoň jedno takové místo. Díky tomu můžeme vynechat použití SimpleCV a vyhnout se tak problémů s ním spojenými.

5.1.3 Virtualizace

Jediný zkoumaný nástroj pro automatizované testování GUI, který podporoval použití virtualizace, byl program os-autoinst. Pro vytváření a ovládání virtuálních strojů však používal dostupné příkazové rozhraní. Tento přístup není vhodný z hlediska přenositelnosti mezi virtualizačními technologiemi. Existuje však API, které se snaží sjednotit programové použití virtualizačních platforem. Toto API se nazývá *libvirt*[4]. Součástí tohoto projektu je výše zmíněné API, dále unixový démon a nakonec klientský software, programové knihovny. Klientský software je multiplatformní a může být spuštěn i na systému Windows [6]. API libvirt obsahuje podporu pro nejpoužívanější virtualizační nástroje, jmenovitě pro QEMU/KVM, VirtualBox, VMware server, OpenVZ, Microsoft Hyper-V a další [5].

Knihovna libvirt přijímá definici VM[5] ve formátu XML a poskytuje funkce pro jejich vytváření a správu. Dle názvosloví knihovny libvirt se jeden virtualizovaný systém nazývá *doména*. Domény jsou spuštěny pomocí hypervizoru, virtualizačních technologií, na určitém *uzlu*. Uzel je fyzický stroj, který je schopný virtualizace. Připojení knihovny libvirt k hypervizoru se řídí nastavením URI, které nese informaci o použité virtualizační technologii. Tento způsob umožňuje, aby hypervizor běžel na jiném stroji, než je spouštěn klientský software. Struktura virtualizačních nástrojů z pohledu knihovny libvirt je na obrázku 5.2.

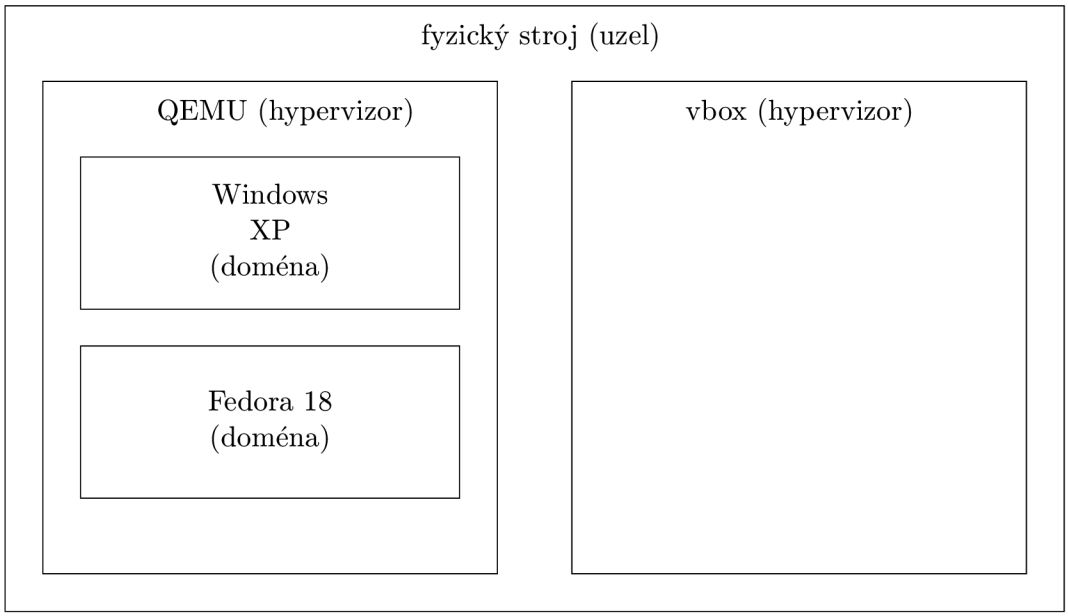
5.1.4 Framework unittest

Protože implementovaný program má za úkol spouštět nezávislé testovací případy, nabízí se možnost využít nějaký ze standardních testovacích frameworků. Framework unittest pro jazyk Python vznikl původně z projektu jménem *PyUnit*. Z jeho názvu je patrné, že patří do rodiny xUnit testovacích frameworků. Ač by se, podle názvu, mohlo zdát, že se jedná o framework určený pouze pro testování jednotek, je stejně tak vhodný také pro jiné typy testování. Projekt Pyramid jej například používá pro testy integrační [2]. Tento přístup nám přinese výhodu standardizovaného řešení, jako například možnost použití integračních

[3] https://bugs.launchpad.net/xpresser/+bug/1060120

[4] http://libvirt.org/

[5] Virtual machine, virtuální stroj



Obrázek 5.2: Architektura virtualizačních technologií z pohledu knihovny libvirt

serverů, projektů Buildbot[6] nebo Jenkins[7].

Pro použití frameworku unittest v kódu je potřeba vytvořit třídy, které dědí od třídy `unittest.TestCase`. Tyto třídy definují dvě speciální metody, `setUp()` a `tearDown()`. Všechny ostatní metody jsou brány jako testovací případy. Framework unittest vytvoří instance těchto tříd a každou jejich metodu zavolá jako samostatný test. Před všemi těmito metodami zavolá metodu `setUp()`. Po každé této metodě zavolá metodu `tearDown()`.

Těla metod tříd, odvozených od třídy `unittest.TestCase` budou samotné spouštěné testovací případy. Nenalezení očekávaného snímku v grafickém výstupu virtualizovaného systému vyústí v zavolání výjimky. Jakékoliv neošetřené zavolání výjimky považuje framework unittest za selhaný test. Další možností je volání metody `unittest.TestCase.fail()`, která okamžitě ukončí aktuálně prováděný testovací případ.

5.2 Dynamické vytváření testovacích případů

Tradiční způsob GUI testování zachycení a přehrání má několik nevýhod. Mezi ty hlavní patří větší nároky na údržbu. S každou novou verzí testovaného programu je nutné celou testovací sadu projít a reflektovat do kódu změny. Některé průchody programem se s novou verzí mohou stát nefunkční, nové mohou vzniknout. Protože instalátor Anaconda linuxové distribuce Fedora prochází změnou téměř s každým vydáním nové verze Fedory, bylo by pracné vytvářet vždy celé nové testy. Další nevýhodou je potřeba vytvářet velké množství

[6] http://trac.buildbot.net/
[7] http://jenkins-ci.org/

testů pro každé kombinace voleb testovaného programu.

Výše uvedenému jsme se snažili vyhnout za pomocí struktury, kterou budeme nazývat „fragment“. Fragment se skládá z několika částí. Tou hlavní částí je testovací funkce. Každý fragment by se měl zaměřit na testování přísně vyhraněné, konkrétní činnosti testovaného programu. Každé takové testovací funkci odpovídá otestování jedné fáze z podkapitoly 4.2. Další součástí fragmentu je seznam závislostí. U každého fragmentu můžeme definovat jiné fragmenty, které jsou pro spuštění testu tohoto fragmentu potřeba, nebo naopak - fragmenty, které se v jednom průchodu testování programu nemohou vyskytnout společně s tímto fragmentem. Volitelnou částí fragmentu je funkce, která na nainstalovaném systému otestuje správnost fungování voleb nastavených před instalací.

Pomocí těchto informací je pak možno získat všechny možné průchody instalací distribuce Fedora a proto výsledné spouštěné testy generovat dynamicky. Problém však nastává při integraci s frameworkem unittest. Jak již bylo řečeno, každý testovací případ musí být metodou třídy. V našem projektu je však nutné vytvářet testovací případy dynamicky. V tradičních, staticky kompilovaných jazycích by šlo o velmi složitý úkol. My však můžeme využít dynamičnosti Pythonu. Ten totiž obsahuje vestavěnou funkci `setattr`, která dokáže nastavit metodu existující třídě. Pseudokód pro dynamické vytváření testovacích případů je následující:

```
1  class InfinityTestCase(unittest.TestCase):
2      def setUp(self):
3          self.virtualization = virtualize()
4
5      def tearDown(self):
6          self.virtualization.destroy()
7
8  def wrapper_function(test_case):
9      def method(self):
10         for fragment in test_case:
11             fragment.run()
12
13     return method
14
15 all_tests = Infinity.generate_all_test_cases()
16 i = 0
17 for test_case in all_tests:
18     setattr(InfinityTestCase, "test"+str(i),
19             wrapper_function(test_case))
```

Tento kód může působit složitě, využívá však velmi jednoduchých principů. Nejprve je vytvořena třída, dědící od `unittest.TestCase`, která definuje pouze metody `setUp()` a `tearDown()`. Funkcí `wrapper_function()` vytvoříme takzvaný *uzávěr* (anglicky *closure*). Zjednodušeně, vnitřní funkce jménem `method()` získá přístup k lokálním proměnným a argumentům nadřazené funkce, zde seznamu fragmentů, neboli testovacímu případu. Výše uvedený zdrojový kód poté vygeneruje všechny možné průchody instalací a použije funkci `setattr` pro vytvoření nových metod existující třídy.

Důležitou součástí programu Infinity je také kód, který generuje testovací případy ze struktury fragmentů. Komplikujícím prvkem je řešení závislostí. Výsledný program proto vygeneruje všechny možné kombinace, pro každou fázi z podkapitoly 4.2 požadovaný počet

fragmentů - testů. Následně program projde touto strukturou a odstraní z ní všechny takové kombinace, ve kterých nejsou splněny požadované závislosti. Použitím struktury fragmentů se nevyhneme nutnosti upravovat testy při každé nové verzi zcela, ale rozdělení do menších, oddělených celků nám zajistí snazší nalezení místa, které je potřeba upravit a také již nebude nutné upravovat celé testovací případy - upravit stačí pouze fragment, ze kterého budou testy generovány.

5.3 Výstup programu a záznam testů

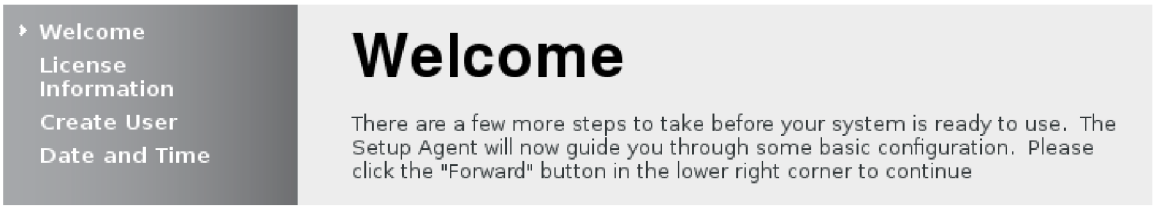
Důležitou vlastností testování je jeho výstup. Původně uvažovaná knihovna *logging* není dostačující, protože je netriviálním úkolem zaznamenávat výstup programu do různých souborů při jednom spuštění programu. Proto bylo pro program Infinity vyvinuto nad knihovnou logging rozšíření. Veškerý výstup z jednoho spuštění programu se zaznamenává do souboru, uloženého v adresáři `complete_logs`, který je označen přesným časem spuštění programu. Pouze chybová hlášení jednotlivých testů se zároveň ukládají do složky, která je označena stejně, jako soubor s kompletními záznamy. Pro každý selhaný test je uloženo, který fragment selhal. Pokud k selhání došlo z důvodu nenalezení požadovaného snímku v testovaném systému, je přiložen název souboru, ve kterém se očekávaný snímek nachází a také snímek stavu, ve kterém testovaný systém byl, když k selhání došlo. Ukázka výstupu je vidět na příkladu 1. Očekávaný snímek je na obrázku 5.3, poslední snímek, který byl na obrazovce nalezen, je na obrázku 5.4.

Příklad 1 Příklad záznamu o selhaném testu

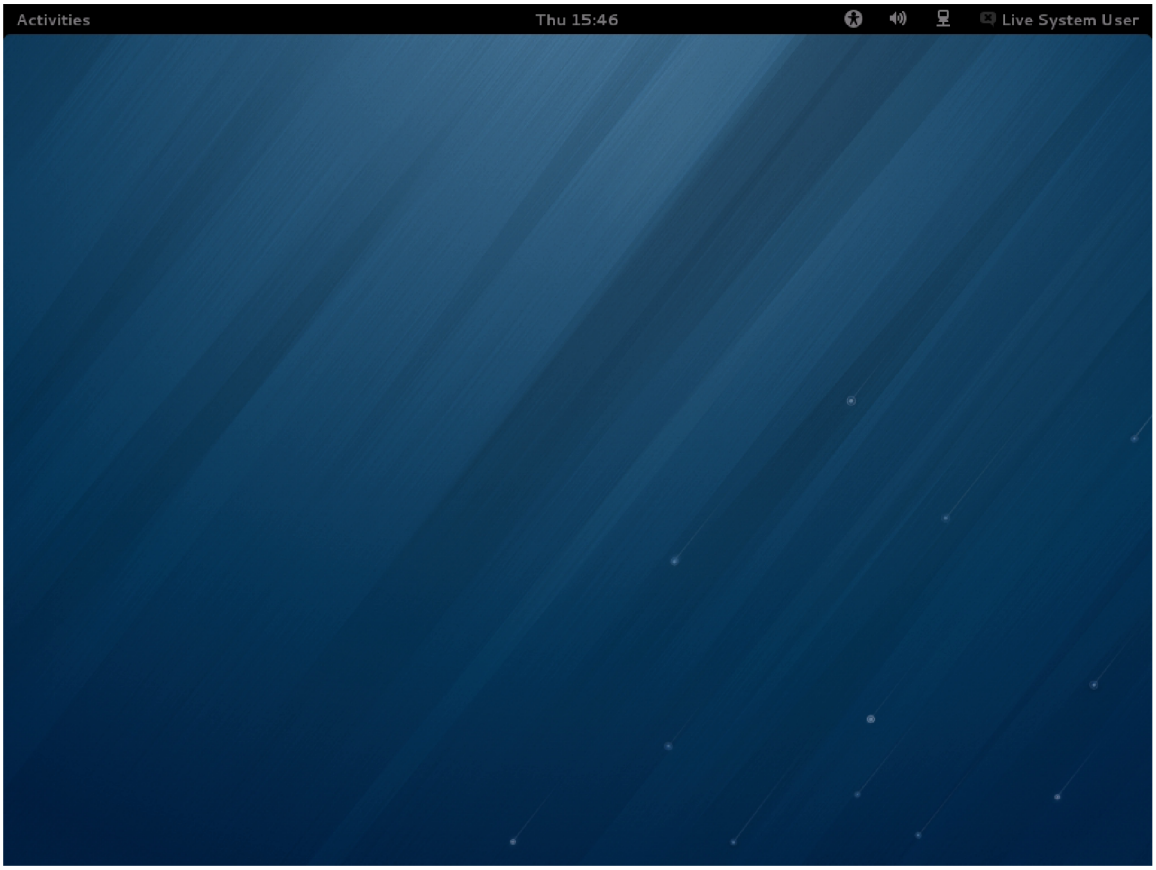
```
2013-05-02 21:33:24:INFO: 0:00:00: starting test 6
2013-05-02 21:33:24:INFO: 0:00:00: stage boot:
2013-05-02 21:33:24:INFO: 0:00:00: fragment boot_installer
2013-05-02 21:34:22:INFO: 0:00:58: fragment lang_default
2013-05-02 21:34:29:INFO: 0:01:05: stage voluntary:
2013-05-02 21:34:29:INFO: 0:01:05: fragment different_timezone
2013-05-02 21:34:37:INFO: 0:01:13: stage part:
2013-05-02 21:34:37:INFO: 0:01:13: fragment part_password
2013-05-02 21:34:50:INFO: 0:01:26: stage install:
2013-05-02 21:34:50:INFO: 0:01:26: fragment root_installation
2013-05-02 21:46:30:INFO: 0:13:05: stage reboot:
2013-05-02 21:46:30:INFO: 0:13:05: fragment reboot_gnome_password
2013-05-02 21:48:11:ERROR: 0:14:48: fragment reboot_gnome_password failed:
 expected .../firstboot.png,
 found .../failed_reboot_gnome_password_DQKh81.png
```

Dalším užitečným výstupem automatizovaného testování GUI, který byl pro program Infinity implementován, je videozáznam. Program Infinity používá snímků získaných pro ovládání UI pro vytvoření videosouboru. Při dlouhých pasážích, jako je například průběh instalace, je možné vytváření videozáznamu dočasně pozastavit. Pro vytváření videozáznamů je používaná knihovna OpenCV a její třída `VideoWriter`. Nahrávání průběhu testů není bohužel dostupné při spouštění na systému Fedora kvůli chybě v balíčku opencv[8].

[8] https://bugzilla.redhat.com/show_bug.cgi?id=812628



Obrázek 5.3: Očekávaná část snímku obrazovky programu firstboot



Obrázek 5.4: Nalezený neočekávaný snímek

5.4 Kritérium pokrytí implementované testovací sady

Sada testuje, zda lze operační systém Fedora bez chyb nainstalovat při výchozím nastavení všech voleb, stejně tak při různém, často používaném dalším nastavení. Navržená testovací sada implementuje test těchto aktivit programu Anaconda:

- Načítání live operačního systému a spuštění programu Anaconda.
- Volba implicitního jazyka. Zde program Anaconda poskytuje volbu jiného jazyka instalace, tato volba zůstává neotestována.
- Nastavení časové zóny. V testovací sadě není implementován test nastavení síťového času, protože se nejedná o kritickou vlastnost. Ve výchozím nastavení je síťový čas zapnut.
- Změna voleb klávesnice, mezi které patří přidání nového rozložení, změnu priority rozložení klávesnice, odebrání rozložení klávesnice a změna nastavení přepínání klávesnice.
- Volby rozdělení disku. Program Anaconda poskytuje možnost automatického rozdělení disku, dále možnost ručního rozdělení disku a možnost disk šifrovat. Poslední volbou nástroje pro rozdělení disku je tzv. *reclaim dialog*, který slouží pro nastavení rozložení disku, pokud na disku již existuje jiný operační systém. Tato volba je však neotestována. Neotestována je také volba primárního disku při instalaci systému Fedora, pokud má počítač více pevných disků.
- Po začátku instalace je nutné nastavit heslo administrátorského účtu.

Po dokončení instalace již končí činnost programu Anaconda a live operační systém je nutno restartovat do nově nainstalovaného operačního systému. Po prvním spuštění systému se zároveň spustí program firstboot, ve kterém je možno vytvořit nové uživatele. I když se již nejedná o testování programu Anaconda, k testování instalace systému Fedora toto patří a proto jsou volby programu firstboot zahrnuty do voleb, které byly otestovány. Jedná se o volby vytvoření nového uživatele a také reakci programu firstboot při vytvoření nového uživatele s diakritikou ve jméně.

Kapitola 6

Závěr

V této práci byly představeny principy testování, principy automatizovaného testování grafického uživatelského rozhraní, dále prozkoumány dostupné nástroje pro automatizované testování GUI s otevřeným zdrojovým kódem. Jeden z těchto nástrojů byl na základě definovaných kritérií vybrán, upraven a použit pro návrh a implementaci testovací sady pro instalaci linuxové distribuce Fedora programem Anaconda. Z důvodu velkého množství možných testovacích případů byla navržena struktura jejich dynamického vytváření z menších částí a tato struktura byla následně vytvořena za pomocí testovacího frameworku unittest.

V rámci praktické implementace navržené testovací sady bylo otestováno 56 případů, každý značí instalaci systému Fedora s jiným nastavením. Průběh těchto testů byl zaznamenán a byly nalezeny dvě chyby integrace programu Anaconda do zbytku systému, jedna již nahlášená[1] a druhá, kterou bylo nutno nahlásit[2].

6.1 Vyhodnocení pokrytí testovací sadou

Testovací sada pokrývá množinu nejčastěji prováděných úkonů při instalaci linuxové distribuce Fedora z výchozího instalačního média. Byly do ní zahrnuty i pokročilejší úkony které zahrnují mimo jiné například nastavení šifrování disku a ruční rozdělení disku či některé hraniční případy, jako je vytvoření uživatele s diakritikou ve jméně. Pro některé případy je zkontrolováno promítnutí nastavených voleb do nainstalovaného systému. Implementovaná testovací sada spolu s použitým upraveným systémem pro automatizované testování GUI by mohly být použity jako poslední krok integračního testování, systémové testování či pro regresní testování programu Anaconda. Použití testovacího frameworku unittest zaručuje jeho snadné použití v systémech pro průběžnou integraci.

Dle kritéria popsaného v kapitole 4 a podkapitole 5.4 bylo implementováno 11 z 15 možných základních voleb programu Anaconda a navíc byla otestována základní funkčnost programu firstboot. Testovací sada nepokrývá z výše uvedených důvodů instalaci v jiném než implicitním jazyce, dále použití jiného než výchozího instalačního média. Z důvodů využití virtualizace při testování nebyly otestovány různé hardwarově-specifické případy jako je použití více disků a testování na širším spektru hardware. Dále nepokrývá instalaci souběžně s jiným operačním systémem. Sada testů byla implementována pro *pozitivní testování*, čili takové testování, které zkoumá, zda aplikace vykazuje požadované fungování při validních vstupech.

[1] https://bugzilla.redhat.com/show_bug.cgi?id=878433
[2] https://bugzilla.redhat.com/show_bug.cgi?id=959752

6.2 Možnosti pro další rozvoj

Implementovaný systém pro automatizované testování GUI jménem Infinity (jehož zdrojové kódy byly zveřejněny na serveru Github[3]) by bylo vhodné předělat do formy knihovny jazyka Python, aby bylo možné použít jej pro testování GUI obecně kterékoliv aplikace. Pro ovládání virtualizovaného operačního systému by bylo dobré použít protokol SPICE, avšak pro tento účel by bylo nutné vytvořit knihovnu pro komunikaci pomocí tohoto protokolu v jazyce Python.

Samozřejmou možností pro další rozvoj testovací sady je implementace takových případů, které testovací sada nepokrývá. Testovací sadu by bylo dále vhodné rozšířit o testy v rámci *negativního testování*, čili takového testování, které má za cíl zkontrolovat chování programu na nevalidní a jinak chybný vstup.

[3] https://github.com/garretraziel/infinity

Literatura

[1] Global Functions and Features [online]. http://doc.sikuli.org/globals.html, 2012-11-08 [cit. 2013-04-27].

[2] Unit, Integration, and Functional Testing [online]. http://docs.pylonsproject.org/projects/pyramid/en/1.3-branch/narr/testing.html, 2013-01-05 [cit. 2013-05-02].

[3] Automating UI Testing [online]. http://developer.apple.com/library/ios/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UsingtheAutomationInstrument/UsingtheAutomationInstrument.html, 2013-04-23 [cit. 2013-05-06].

[4] Platforms Supported by Selenium [online]. http://docs.seleniumhq.org/about/platforms.jsp#programming-languages, [cit. 2013-04-27].

[5] Internal drivers [online]. http://libvirt.org/drivers.html, [cit. 2013-05-02].

[6] Windows support [online]. http://libvirt.org/windows.html, [cit. 2013-05-02].

[7] UI Testing [online]. http://developer.android.com/tools/testing/testing_ui.html, [cit. 2013-05-04].

[8] Ammann, P.; Offutt, J.: *Introduction to software testing*. Cambridge University Press, 2008, ISBN 13 978-0-511-39330-3.

[9] Apple Inc.: Well known TCP and UDP ports used by Apple software products [online]. https://support.apple.com/kb/TS1629, 2012-11-13 [cit. 2013-05-02].

[10] Beck, K.: *Programování řízené testy*. Grada Publishing, a.s., 2004, ISBN 80-247-0901-5.

[11] Burnstein, I.: *Practical Software Testing: a process-oriented approach*. Springer-Verlag New York, Inc., 2003, ISBN 0-387-95131-8.

[12] Chang, T.-H.: Sikuli-vnc [online]. https://code.launchpad.net/~sikuli-vnc/sikuli/sikuli-vnc, 2011-01-12 [cit. 2013-04-27].

[13] Chang, T.-H.; Yeh, T.; Miller, R.: GUI Testing Using Computer Vision. http://groups.csail.mit.edu/uid/projects/sikuli/sikuli-chi2010.pdf.

[14] Dustin, E.; Rashka, J.; Paul, J.: *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999, ISBN 0-201-43287-0.

[15] George, B.; Williams, L.: A structured experiment of test-driven development. *Information and Software Technology*, ročník 46, č. 5, 2004: s. 337–342.

[16] Jorgensen, P. C.: *Software testing: A Craftsman's Approach*. Auerbach Publications, 2008, ISBN 978-0-8493-7475-3.

[17] Kaner, C.; Falk, J.; Nguyen, H. Q.: *Testing Computer Software Second Edition*. John Wiley & Sons, 2000, ISBN 0-471-35846-0.

[18] Kasik, D. J.; George, H. G.: Toward automatic generation of novice user test scripts. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 1996, s. 244–251.

[19] Li, K.; Wu, M.: *Effective GUI testing automation: Developing an automated GUI testing tool*. Sybex, 2006.

[20] Marick, B.: *The craft of software testing: subsystem testing including object-based and object-oriented testing*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995, ISBN 0-13-177411-5.

[21] Memon, A. M.; Pollack, M. E.; Soffa, M. L.: Using a goal-driven approach to generate test cases for GUIs. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, IEEE, 1999, s. 257–266.

[22] Microsoft Corporation: Test an Application Using Capture and Playback [online]. http://msdn.microsoft.com/en-US/library/ee253074%28v=bts.10%29.aspx, 2004 [cit. 2013-05-05].

[23] Microsoft Corporation: Pex and Moles - Isolation and White box Unit Testing for .NET [online]. http://research.microsoft.com/en-us/projects/pex/, [cit. 2013-05-08].

[24] Pezzè, M.; Young, M.: *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons Inc, 2008, ISBN 13 978-0-471-45593-6.

[25] Pilgrim, M.: *Ponořme se do Python(u) 3*. CZ.NIC, z. s. p. o., 2010, ISBN 978-80-904248-2-1.

[26] Rushby, J.: Automated test generation and verified software. In *Verified Software: Theories, Tools, Experiments*, Springer, 2008, s. 161–172.

[27] Testing Standards Working Party: Working Draft of BS 7925-1 [online]. http://www.testingstandards.co.uk/Gloss6_3.zip, 2004-09-06 [cit. 2013-04-22].

[28] Yeh, T.; Chang, T.-H.; Miller, R.: Sikuli: Using GUI Screenshots for Search and Automation. http://groups.csail.mit.edu/uid/projects/sikuli/sikuli-uist2009.pdf.

Příloha A

Příklady použití nástrojů na testování GUI

A.1 os-autoinst

Tento příklad je převzat ze zdrojových kódů programu os-autoinst. Jedná se o text funkčnosti programu Firefox. Program byl upraven a zkrácen, slouží pouze pro ilustraci.

```
1  use base "basetest";
2  use bmwqemu;
3
4  sub run()
5  {
6          my $self=shift;
7          mouse_hide(1);
8          x11_start_program("firefox");
9          $self->take_screenshot;
10         if($ENV{UPGRADE}) { sendkey("alt-d");waitidle; }
11         if($ENV{DESKTOP}=~/xfce|lxde/i) {
12                 sendkey "ret";
13                 waitidle;
14         }
15         if(0) {
16                 sendkey "ctrl-t"; sleep 1;
17                 sendautotype "about:config\n"; sleep 1;
18                 sendkey "ret"; waitidle;
19                 sendautotype "showQuit\n\t"; sleep 1;
20                 sendkey "ret"; waitidle;
21                 sendkey "ctrl-w"; sleep 1;
22         }
23         $self->take_screenshot;
24         sendkey "alt-h"; sleep 2;
25         sendkey "a"; sleep 2;
26         $self->take_screenshot;
27         sendkey "alt-f4"; sleep 2;
28         sendkey "alt-f4"; sleep 2;
```

```
29          sendkey "ret";
30  }
31  sub checklist()
32  {
33          return {qw(
34                  fc382651f1e1b6359789038ad0bd9bc0 OK
35                  4299006210d21ee52570d99916500f76 OK
36                  a10028c503d80f78603f4bd79cbab29d OK
37                  c7bcf2e6976800803da351d4e6108fdb fail
38          )}
39  }
40  1;
```

A.2 Autopilot

Tento kód je testem ze zdrojových kódů projektu Unity. Slouží k testování prvku *Home Lens*. Příklad byl zkrácen a upraven.

```
1  from __future__ import absolute_import
2  from autopilot.matchers import Eventually
3  from testtools.matchers import Equals
4  from time import sleep
5  from unity.tests import UnityTestCase
6
7  class HomeLensSearchTests(UnityTestCase):
8      def setUp(self):
9          super(HomeLensSearchTests, self).setUp()
10
11     def tearDown(self):
12         self.unity.dash.ensure_hidden()
13         super(HomeLensSearchTests, self).tearDown()
14
15     def test_quick_run_app(self):
16         if self.app_is_running("Text Editor"):
17             self.close_all_app("Text Editor")
18             sleep(1)
19
20         kb = self.keyboard
21         self.unity.dash.ensure_visible()
22         kb.type("g")
23         self.assertThat(self.unity.dash.search_string,
24                         Eventually(Equals("g")))
25         kb.type("edit", 0.1)
26         kb.press_and_release("Enter", 0.1)
27         self.addCleanup(self.close_all_app, "Text Editor")
28         app_found = self.bamf.wait_until_application_is_running(
29                                                 "gedit.desktop", 5)
30         self.assertTrue(app_found)
```

A.3 Dogtail

Příklad testovacího skriptu ze zdrojových kódů programu Dogtail.

```
1  from dogtail.tree import *
2  from dogtail.utils import run
3  from sys import exit
4
5  run('gedit')
6  gedit = root.application('gedit')
7
8  while True:
9      try:
10         gedit.child('Open').click()
11     except SearchError: #toolbar not present?
12         gedit.child('Open...').click()
13
14     try:
15         filechooser = gedit.child(name='Open␣Files',
16                                   roleName='file␣chooser')
17         filechooser.childNamed('Cancel').click()
18     except SearchError:
19         print('File␣chooser␣did␣not␣open')
20         exit(1)
```

A.4 Xpresser

Příklad testovacího skriptu, který používá knihovnu Xpresser.

```
1  from xpresser import Xpresser
2  import unittest
3
4  class FirefoxTestCase(unittest.TestCase):
5      def setUp(self):
6          self.xp = Xpresser()
7          self.xp.load_images("images")
8
9      def test_firefox_search(self):
10         self.xp.click("firefox-button")
11         self.xp.wait("firefox-opened", timeout=5)
12         self.xp.click("firefox-search")
13         self.xp.type("xpresser")
14         self.xp.click("firefox-searched")
15         self.xp.wait("firefox-google")
```

Příloha B

Postup instalace a spuštění implementovaného nástroje na linuxové distribuci Fedora

Program Infinity vyžaduje ke svému běhu počítač s alespoň 2 GiB operační paměti RAM a 64bitovým procesorem s podporou virtualizace (AMD-V nebo Intel VT). Byl otestován na linuxových distribucích Arch Linux a Fedora. Na operačním systému Fedora verze 18 je postup instalace a spuštění následující:

1. Nainstalujte programy QEMU s podporou KVM a knihovny libvirt a OpenCV s podporou programovacího jazyka Python následujícím příkazem:

   ```
   sudo yum install qemu-kvm libvirt-python opencv-python
   ```

2. Můžete nastavit práva pro používání knihovny libvirt dle tohoto návodu: https://wiki.archlinux.org/index.php/Libvirt#PolicyKit_authorization. Jinak lze výsledný program spouštět pod uživatelem *root*.

3. Zkopírujte souborovou strukturu projektu do Vaší domovské složky.

4. Z oficiálních webových stránek projektu Fedora[1] stáhněte instalační obraz distribuce Fedora 18, 64bitovou verzi. Stahovaný soubor by se měl jmenovat `Fedora-18-x86_64-Live-Desktop.iso`.

5. Uložte tento obraz do adresářové struktury projektu do složky `resources/machines` pod názvem `image-live.iso`.

6. Spusťte program Infinity příkazem `python infinity.py`. Výstup programu se bude nacházet ve složce `logs`.

Pokud Váš systém používá jinou implicitní verzi interpretru programovacího jazyka Python než je verze 2 (například linuxová distribuce Arch Linux), je nutné nastavit interpretr pro Python 2 jako implicitní. K tomu lze použít například nástroj *virtualenv*[2].

[1] https://fedoraproject.org/
[2] http://www.virtualenv.org/en/latest/

Příloha C

Testování nároků programu Anaconda

Implementovaný nástroj byl také využit při testování nároků programu Anaconda na operační paměť. Virtuálnímu stroji byla nastavována různá velikost paměti RAM a metodou bisekce byla nalezena hodnota její minimální dostatečné velikosti. Pseudokód způsobu hledání této hodnoty je následující:

```
1  import xpresser, fragments, virtualisation
2
3  low = 128 # pri 128 MiB urcite selze
4  high = 1024 # pri 1 GiB projde
5  interval = 1
6  center = (high-low)/2
7
8  while center > interval:
9    memory = low+center
10   vm = create_vm(ram=memory)
11   failed = False
12   for fragment in fragments:
13     try:
14       fragment.run()
15     except FragmentFailed:
16       failed = True
17       break
18   if failed:
19     low = memory
20   else:
21     high = memory
22   center = (high-low)/2
23
24 print "Anaconda vyzaduje alespon", center+low, "MiB RAM."
```

Výsledný skript je nazván `ramtest.py` a je přiložen ke zdrojovým kódům. Jeho spuštěním bylo zjištěno, že program Anaconda potřebuje ke svému běhu minimálně 635 MiB RAM. Tato hodnota je však pouze orientační. Je závislá na mnoha faktorech, a proto může vycházet různě na různých počítačích.