FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Diplomová práce

Generování jednotkových testů s využitím LLM

Milan Horínek

PLZEŇ

2024

FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Diplomová práce

Generování jednotkových testů s využitím LLM

Bc. Milan Horínek

Vedoucí práce
Ing. Richard Lipka, Ph.D.

PLZEŇ **2024**

Citace v seznamu literatury:
HORÍNEK, Milan. *Generování jednotkových testů s využitím LLM*. Plzeň, 2024. Diplomová práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky. Vedoucí práce Ing. Richard Lipka, Ph.D.

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2023/2024

Studijní program: Softwarové a informační systémy
Forma studia: Prezenční

Podklad pro zadání DIPLOMOVÉ práce studenta

Jméno a příjmení:	**Bc. Milan HORÍNEK**
Osobní číslo:	**A23N0089P**
Adresa:	Česká 1024/6, Most, 43401 Most 1, Česká republika
Téma práce:	Generování jednotkových testů s využitím LLM
Téma práce anglicky:	LLM based unit test generator
Jazyk práce:	Čeština
Vedoucí práce:	Ing. Richard Lipka, Ph.D. Katedra informatiky a výpočetní techniky

Zásady pro vypracování:

1. Seznamte se s existujícími LLM, jejich technologií a možnostmi použití.
2. Prostudujte exitující literaturu ohledně genrování jednotkových testů, zejména s ohledem na využití neuronových sítí a LLM.
3. Seznamte se s existujícími ukázkovými programy s možností injekce chyb a vyberte vhodný testovací program.
4. Navrhněte a implementujte automatizovaný nástroj využívající popis testu v přirozené řeči, LLM a případně další informace, který vygeneruje sadu jednotkových testů.
5. Ověřte kvalitu automaticky vytvořených testů zejména s ohledem na přesnost a úplnost.
6. Zhodnoťte možnosti současných LLM pro generování testů.

Seznam doporučené literatury:

Dodá vedoucí práce.

Podpis studenta: Datum:

Podpis vedoucího práce: Datum:

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného akademického titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Západočeská univerzita v Plzni má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

Plzeň dne 14. května 2024

. . . . . .
Milan Horínek

V textu jsou použity názvy produktů, technologií, služeb, aplikací, společností apod., které mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

Abstrakt

TBD

Abstract

TBD

Klíčová slova

LLM • testing • unit • Robot Framework

Poděkování

TBD

Obsah

Úvod 1

TDB - Co jsou unit testy, LLM, a jejich předpoklady, GUI testování

Něco na úvod

1.1 Testy

1.2 Jazykové modely

1.3 Motivace

1.4 Co a jak testujeme

1.5 Navrhované řešení

Rešerše 2

2.1 Provedená práce v problematice

2.1.1 Předchozí automatizovaná řešení

Jazykové modely nebyli prvními pokusy o automatizované generování jednotkových testů. Ještě před nimi existovala spousta metod zahrnující příklady jako *fuzzing, generování náhodných testů řízených zpětnou vazbou, dynamické symbolické exekuce, vyhledávvací a evoluční techniky, parametrické testování*. Zároveň také již na počátku století byli pokusy o vytvoření vlastní neuronové sítě sloužící právě čistě k úkolu testování softwaru. V této sekci je ukázka několika z nich.

2.1.1.1 Programatická řešení

Jedna z používaných programatických automatizovaných metod pro tvrobu jednotkových testů je tzv. *fuzzing*. V rámci těchto testů musí uživatel stále definovat jeho kódovou strukturu, resp. akce, které test bude provádět a jaký výstup očekávat. Automaticky generovaný je pouze vstup tohoto testu. Výhodou zde tedy je, že uživatel nemusí vytvářet maketu vstupních dat testu, která se zde vytvoří automatizovaně. Zůstává zde však problematika, že pro uživatele není kód *black-box*, ale celou jeho strukturu včetně požadovaného výstupu musí sám definovat. [SGA07]

Pouze vstupy dokáže také generovat metoda *symbolické exekuce*, která postupně analyzuje chování větvení programu. Začíná bez předchozích znalostí a používá řešitel omezení k nalezení vstupů, které prozkoumají nové exekuční cesty. Jakmile jsou testy spuštěny s těmito vstupy, nástroj sleduje cestu, kterou se program ubírá, a aktualizuje svou znalostní bázi (q) s novými podmínkami cesty (p). Tento iterativní

proces se opakuje a nadále zpřesňuje sadu známých chování a snaží se maximalizovat pokrytí kódu. Nástroje běžně zvládají různé datové typy a respektují pravidla viditelnosti objektů. Snaží se také používat mock objekty a parametrizované makety k simulaci různých chování vstupů, čímž zlepšuje proces generování testů, aby odhalila potenciální chyby a zajistila komplexní pokrytí kódu testy. [Par] Tato metoda je implementována například v nástroji IntelliTest v rámci IDE *Visual Studio*. Je používáná v kombinaci s *parametrickými testy*, také označovanými jako PUT. Ty na rozdíl od tradičních jednotkových testů, které jsou obvykle uzavřené metody, mohou přijímat libovolnou sadu parametrů. Nástroje se pak snaží automaticky generovat (minimální) sadu vstupů, které plně pokryjí kód dosažitelný z testu. Nástroje jako např. *IntelliTest* automaticky generují vstupy pro put, které pokrývají mnoho exekučních cest testovaného kódu. Každý vstup, který pokrývá jinou cestu, je "serializován"jako jednotkový test. Parametrické testy mohou být také generické metody, v tom případě musí uživatel specifikovat typy použité k instanci metody. Testy také mohou obsahovat atributy pro očekávané a neočekávané vyjímky. Neočekávané vyjímky vedou k selhání testu. put tedy do velké míry redukují potřebu uživatelského vstupu pro tvorbu jednotkových testů. [Mic23a] [Mic23b]

Pokud zvolíme symbolické řešení vstupu společně s determinovanými vstupy a testovací cestou, vzniká tak hybridní řešení zvané jako *konkolické testovaní* nebo *dynamická symbolická exekuce*. Tento druh testů dokáží tvořit nástroje jako *SAGE, KLEE* nebo *S2E*. Problémem tohoto přístupu však je, když program vykazuje *nedeterministické* chování, kdy tyto metody nebudou schopny určit správnou cestu a zároveň tak ani zaručit dobré pokrytí kódu/větví. Velká míra používání stavových proměnných může vést k vysoké výpočetní náročnosti těchto metod a nenalezení praktického řešení. [EC06] [SMA05] [Zho+06]

Další metodou je *náhodné generování testů řízené zpětnou vazbou*, která je vylepšením pro generování náhodných testů tím, že zahrnuje zpětnou vazbu získanou z provádění testovacích vstupů v průběhu jejich vytváření. Tato technika postupně buduje vstupy tím, že náhodně vybírá metodu volání a hledá argumenty mezi dříve vytvořenými vstupy. Jakmile dojde k sestavená vstupu, je provedena jeho exekuce a výsledek ověřen proti sadě kontraktů a filtry. Výsledek exekuce určuje, zda je vstup redundantní, proti pravidlům, porušující kontrakt nebo užitečný pro generování dalších vstupů. Technika vytváří sadu testů, které se skládají z jednotkových testů pro testované třídy. Úspěšné testy mohou být použity k zajištění faktu, že kódu jsou zachovány napříč změnami programu; selhávající testy (porušující jeden nebo více kontraktů) ukazují na potenciální chyby, které by měly být opraveny. Tato metoda dokáže vytvořit nejen vstup pro test, ale i tělo (kód) testu. Ovšem pro uživatele je stále vhodné znát strukturu kódu. [Pac+07]

Z programatických metod se zdají být nejpokročilejší *evoluční algoritmy* pro generování sad jednotkových testů, využívající přístup založený na vhodnosti, aby vyvíjely testovací případy, které mají za cíl maximalizovat pokrytí kódu a detekci chyb. Tyto algoritmy mohou autonomně generovat testovací vstupy, které jsou navrženy tak, aby prozkoumávaly různé exekuční cesty v aplikaci. Uživatelé mohou interagovat s vygenerovanými testy jakožto s *black-boxem*. Testy se zaměřují na vstupy a výstupy, aniž by potřebovali rozumět vnitřní logice testovaného systému. Tento aspekt evolučního testování je zvláště výhodný při práci se složitými systémy nebo když zdrojový kód není snadno dostupný. Proces iterativně upravuje testovací případy na základě pozorovaných chování, upravuje vstupy pro efektivnější prozkoumání systému a identifikaci potenciálních defektů. Tato metoda podporuje vysokou úroveň automatizace při generování testů, snižuje potřebu manuálního vstupu a umožňuje komplexní pokrytí testů s menším úsilím. [Cam+18] [LKF21]

2.1.1.2 Neuronové sítě

Ke generování jednotkových testů lze využít i vlastní neuronové sítě. Takové se pokoušeli vytvářet například v práci "Unit test generation using machine learning"[Sae18], kde byly testovány primárně RNN a experimentálně CNN sítě (ty však měli problém s větším množstvím tokenů). Modely byli testovány na jazyce Java. Metoda přistupovala k programům jakožto white box, tedy měli k dispozici celý zdrojový kód včetně zkompilovaného bytecodu. Při nejlepší konfiguraci dosáhl výsledek modelu 70.5% parsovatelného kódu (tedy takového bez chyb) natrénovaný z bezmála 10000 příkladů *zdrojový kód - test*. Výsledek práce je však stále jakýsi "proof of concept", protože zatímco vygenerují částečně použitelný výsledek, je vždy nutný zásah experta, aby mohlo dojít k vytvoření celého testovacího souboru. Takovéto sítě se však mohou silně hodit jako výpomoc programátorovi při psaní testů.

2.1.1.3 Nevýhody současných metod

Současné metody generování jednotkových testů, jako je *fuzzing* a *symbolická exekuce*, často vyžadují podrobnou znalost struktury kódu a očekávaných výstupů, což omezuje jejich efektivitu a zvyšuje složitost tvorby testů. Jen některé z těchto metod jsou schopné vygenerovat testy pouze na bázi specifikace (z black box pohledu) bez vnitřní znaloasti kódu. Většina z klasických metod je zároveň schopna generovat pouze vstupy jednotkových testů, ale už ne samotné tělo (kód) testu nebo očekávané výstupy, a tedy pouze složí jako jakási konstra pro programátora, který musí test doimplementovat.

Velké jazykové modely (LLM) mohou být atraktivní alternativou, protože pomocí nich lze potenciálně automatizovat generování jak vstupů pro testy, tak přidruženého testovacího kódu, čímž se snižuje potřeba hlubokého porozumění struktuře kódu. S takovým nástrojem není potřeba programátora, ale může s ním pracovat i méně zkušený uživatel (např. *tester*). Dále je zde možnost otestovat kód za pomocí slovní specifikace pro funkci nebo vlastnosti. Takové specifikace se používají například v *aero-space* nebo *automotive* sektoru. LLM také mohou objevit všechny možné stavy, které mohou u vstupu nebo výstupu nastat a pokusit se pro ně navrhnout test.

2.1.2 Vydané publikace

Jeden z poměrně nedávno vydaných článků (září 2023) nazvaný "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation"[Sch+23] se zabývá využitím velkých jazykových modelů (LLM) pro automatizované generování jednotkových testů v jazyce JavaScript. Implementovali nástroj s názvem **TESTPILOT**, který využívá LLM *gpt3.5-turbo*, *code-cushman-002* od společnosti OpenAI a také model StarCoder, který vznikl jako komunitní projekt [Li+23]. Vstupní sada pro LLM obsahovala signatury funkcí, komentáře k dokumentaci a příklady použití. Nástroj byl vyhodnocen na 25 balíčcích npm obsahujících celkem 1684 funkcí API. Vygenerované testy dosáhly pomocí gpt3.5-turbo mediánu pokrytí příkazů 70,2% a pokrytí větví 52,8%, čímž překonaly nejmodernější techniku generování testů v jazyce JavaScript zaměřenou na zpětnou vazbu, Nessie.

Zmíněný model *StarCoder* byl představen v článku "StarCoder: may the source be with you!"[Li+23] z května 2023. Vytvořeny byly konkrétně 2 verze, *StarCoder* a *StarCoderBase*, s 15,5 miliardami parametrů a délkou kontextu 8K. Tyto modely jsou natrénovány na datové sadě nazvané *The Stack*, která obsahuje 1 bilion tokenů z permisivně licencovaných repozitářů GitHub. *StarCoderBase* je vícejazyčný model, který překonává ostatní modely open-source LLM modely, zatímco *StarCoder* je vyladěná verze speciálně pro Python, která se vyrovná nebo překoná stávající modely zaměřené čistě na Python. Článek poskytuje komplexní hodnocení, které ukazuje, že tyto modely jsou vysoce efektivní v různých úlohách souvisejících s kódem.

Článek "Exploring the Effectiveness of Large Language Models in Generating Unit Tests"[Sid+23] z dubna 2023 hodnotí výkonnost tří LLM - *Codex*, *CodeGen* a *GPT-3.5* - při generování jednotkových testů pro třídy jazyka Java. Studie používá jako vstupní sady dva benchmarky, HumanEval a Evosuite SF110. Klíčová zjištění ukazují, že *Codex* dosáhl více než 80% pokrytí v datové sadě *HumanEval*, ale žádný z modelů nedosáhl více než 2% pokrytí v benchmarku *SF110*. Kromě toho se ve vy-

generovaných testech často objevovaly tzv. testové pachy, jako jsou *duplicitní tvrzení* a *prázdné testy* [21].

2.1.2.1 Srovnání výsledků

Výsledky diskutovaných studií v předchozím bodě jsme srovnali v tabulce 2.1. V rámci první práce dosahuje nejlepších výsledků model *gpt-3.5-turbo*, který dosáhl 70% pokrytí kódu testy a 48% úspěšnosti testů. U druhé studie má tento model na testovací sadě *HumanEval* velice podobný výsledek, ovšem model *Codex* dosáhl lepších výsledků. Může však také jít o rozdíl způsobený programovacím jazykem. Zatímco v práci [Sch+23] se využívá jako benchmark sada balíčků jazyka *JavaScript*, který kvůli absenci explicitního typování, může být obtížnější pro strojové testování oproti jazyku Java, který je využit ve zbylých 2 pracích. [Kat23] také využívá Javu a s modelem *gpt-3.5-turbo* dosahuje podobného pokrytí kódu a úspěšnosti jako *Codex* v práci [Sid+23].

2.1.3 Modely

Na LLM modelech nás konkrétně zajímá schopnost pozorumět progamovacím jazykům a ty poté také generovat na výstupu. Důležité pro nás také je, zda daný model je proprietární či otevřený a pod jakou licencí, tedy zda by byl vhodný pro naši práci. V případě analýzy zdrojového kódu může být také klíčovou vlastností délka kontextu daného modelu. Tyto vlastnosti jsou také zaneseny do tabulky 2.2.

StarCoder. Jedním z často používaných modelů v předchozích pracích je *StarCoder* a *StarCoderBase*, diskutovaný již v sekci 2.1. *Base* verze je schopna generovat kód pro více jak 80 programovacích jazyků. Model je navržen pro širokou škálu aplikací obsahující *generování*, *modifikaci*, *doplňování* a *vysvětlování* kódu. Jeho distribuce je volná a licence **CodeML OpenRAIL-M 0.1** [Pro23] umožňuje ho využívat pro množství aplikací včetně komerčních nebo edukačních. Jeho uživatel však má povinnost uvádět, že výsledný kód byl vygenerován modelem. Licence má své restrikce z obavy tvůrců, protože by model mohl někoho při nepsrávném použití ohrozit. Tyto restrikce se aplikují na na všechny derivace projektů pod touto licencí. Zároveň není kompatibilní s Open-Source licencí právě kvůli těmto restrikcím.

Model StarCoder se také dočkal novější verze *StarCoder2*, který nepřímo nava-

zuje na model *StarCoderBase*. Je naučen na archivu GitHub repoziářů archivovaných v rámci organizace *Software Heritage*, čítajících přes 600 programovacích jazyků a dalších pečlivě vybraných dat jako například *pull requesty*. Mimo toho také trénovací data obsahují staženou dokumentaci k vybraným projektům. Model se vyjímá tím, že se snaží udržet malou velikost. Je nabízen ve verzích s *3, 7* a *15* miliardy parametrů, i přesto však dle jejich úvodní studie [**lozhkov2024starcoder**] nabízí na sadě populárních programovacích jazyků shodné či lepší výsledky oproti modelu *CodeLlama* s 34 miliardy parametrů. Výhodou tohoto modelu nepochybně je, že ho lze spustit na spoustě dnešních počítačů s plným offloaded na grafickou kartu.

CodeLlama. Nedávno vydaným modelem je *Code Llama* od společnosti Meta. Jedná se o evoluci jejich jazykového modelu *Llama* specializovaný však čistě na úlohy kódování. Je postaven na platformě *Llama 2* a existuje ve třech variantách: *základní Code Llama*, *Code Llama - Python* a *Code Llama - Instruct*. Model podporuje více programovacích jazyků, včetně jazyků jako Python, C++, Java, PHP, Typescript, C nebo Bash. Je určen pro úlohy, jako je generování kódu, doplňování kódu a ladění. Code Llama je zdarma pro výzkumné i komerční použití a je uvolněn pod licencí MIT. Uživatelé však musí dodržovat zásady přijatelného použití, ve kterých je uvedeno, že model nelze použít k vytvoření služby, která by konkurovala vlastním službám společnosti Meta.

Copilot. Velmi populárním nástrojem pro generování kódu za pomocí LLM je GitHub Copilot, který je postaven na modelu *codex* od OpenAI. Původní model však byl přestal být zákazníkům nabízen a namísto něj OpenAI doporučuje ke generování kódu využívat chat verze modelů GPT-3.5 a GPT-4. Na architektuře GPT-4 je také postavený nástupce služby Copilot, Copilot X. Zmíněné modely chat GPT-3.5 a GPT-4 jsou primárně určeny pro generování textu formou chatu. Zvládají však zároveň i dobře generovat kód a jsou vhodné i úlohu generování jednotkových testů. Narozdíl od předchozích modelů však nejsou volně distribuovány a jsou poskytovány pouze jako služba společností OpenAI skrze API nebo je lze hostovat v rámci služby Azure společnosti Microsoft, která zajišťuje větší integritu dat. Jedná se tedy o uzavřený model a jeho uživatelé musí souhlasit s jeho podmínkami použití.

Práce	Model	Benchmark	Pokrytí testy	Úspěšnost
An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation	*gpt-3.5-turbo*	Sada NPM balíčků	70.2%	48%
	code-cushman-002		68.2%	47.1%
	StarCoder		54%	31.5%
Exploring the Effectiveness of Large Language Models in Generating Unit Tests	*gpt-3.5-turbo*	HumanEval	69.1%	52.3%
		SF110	0.1%	6.9%
	CodeGen	HumanEval	58.2%	23.9%
		SF110	0.5%	30.2%
	Codex (4k)	HumanEval	87.7%	76.7%
		SF110	1.8%	41.1%
Java Unit Testing with AI: An AI-Driven Prototype for Unit Test Generation	*gpt-3.5-turbo*	JUTAI - Zero-shot, temperature: 0	84.7%	71%

Tabulka 2.1: Přehled a srovnání studií

Model	Otevřenost	Licence	Počet parametrů	Počet programovacích jazyků	Datum vydání
StarCoderBase	Volně dostupný	CodeML OpenRAIL-M 0.1	15.5 miliardy	80+	5/2023
Code Llama	Volně dostupný	Llama 2 Licence	34 miliard	8+	8/2023
gpt-3.5-turbo	Uzavřený	Proprietární	175 miliard?	?	5/2023
gpt-4	Uzavřený	Proprietární	1.76 bilionu	?	3/2023

Tabulka 2.2: Přehled a srovnání modelů generujících kód

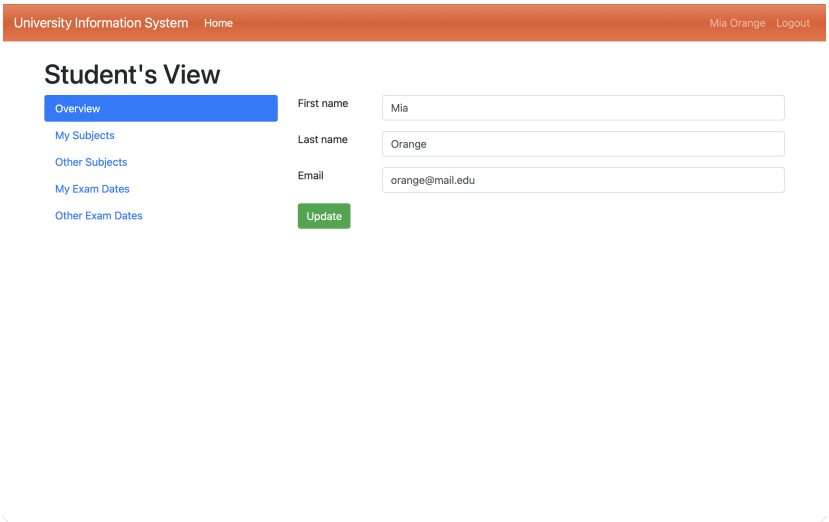
2.2 Testovací program

Pro účely této práce jsme se rozhodli vydat cestou GUI testů, konkrétně webových stránek / webové aplikace. Mezi požadavky na testovanou aplikaci a její výběr bylo mimo možnosti vytvořit pro ní automatiovaně sadu testů, také možnost zavedení (*injekce*) chyb do aplikace, díky kterým bude možno ověřit nejen fakt, že vygenerované testy jsou schopné detekovat *korektní* chování softwaru, ale také úspěšně detekovat *chyby*. Volitelnou podmínkou také byla existence již existujících testů vytvořených lidským programátorem, se kterými by bylo možné strojově generované testy porovnat.

Těmto požadavkům vyhovuje jeden z předešlých univerzitních projektů nazvaný **TbUIS** (*Testbed University Information System*), na který vytvořili Matyáš a Šmaus pod vedením doc. Herouta. [Mat18] [Šma19] Aplikace představuje *maketu* univerzitního informačního systému, avšak i tak implementuje většinu functionality, kterou bysme od podobného systému čekali a nabízí pohled jak ze strany *studenta* tak ze strany *vyučujícího* (viz obr. 2.1). Funkce tohoto systému vycházejí ze sady *use casů*, které popisuje web [1] aplikace. Výhodou tohoto systému primárně je, že nabízí nejen plně funkční a otestovanou variantu, ale také 27 dalších poruchových klonů, vždy porušující alespoň jeden *use case* a s ním potenciálně spojený test.

Projekt dále také obsahuje i sadu jak *funkčních* testů primárně pro backend vytvořených Poubovou tak také *akceptačních* testů držejících se pevně specifikací, vytvořených Vaisem. [Pou19] [Vai20] Tyto testy využívají nejen pevné znalosti jednotlivých uživatelských scénářů, ale také jsou parametrizovány pomocí všech dostupných dat k aplikaci (*např. uživatelská jména, předměty, zkoušky, atd.*) k vyhodnocení chování softwaru v co nejvíce případech. Právě *akceptační* testy využívají námi zvolený **RobotFramework** pro testování a tedy vytvořené scénáře bude možné do určité míry přirovnat a případně zhodnotit jejich kvalitu.

[1] Aplikaci lze nalézt na adrese: `https://projects.kiv.zcu.cz/tbuis/web/`



Obrázek 2.1: Prostředí systému TbUIS z pohledu studenta.

Generování testů 3

3.1 Výběr scénářů

Pro vygenerování testů za pomocí LLM bylo nejdříve nutné vybrat ze sady *use caseů*[1] scénáře vhodné pro demonstraci nejen správné funkčnosti, ale také s možností ověření nesprávného chování na poruchových klonech (diskutováno v sekci 2.2). Požadavek tedy byl, aby většina z vybraných scénářů vycházející z *use casu* měl alespoň jeden poruchový klon, případně více. Vybráno bylo 10 scénářů, testovaných celkově na 14 variantách programu. Seznam vytvořených specifikací pro automatické generování testů se nachází v tabulkách 3.1 a 3.1. Scénáře budou dále také referovány jako *specifikace*. Každá specifikace má *číslo*, které odpovídá číslu *use casu*, ze kterého vychází (např. specifikace 04 vychází z use casu *UC.04*). Zde je nutné poznamenat, že ne všechny specifikace vychází pevně z jejich use casů, ale byli upravené tak, aby šli provést v jednom chodu bez nutných závislostí (*jako například namísto podmínky přihlášeného uživatele se uživatel vždy musí na začátku či během testu přihlásit do systému*). Popis specifikací v tabulce je pouze orientační a pro bližší upřesnění jednotlivých kroků, které se mají provést referujte web projektu. V tomto seznamu se také u každé specifikace nachází výčet klonů, na kterých lze očekávat poruchu. Celkový seznam klonů, a kterých se budou všechny vytvořené testy spouštět lze nalézt v tabulce 3.1. Podobně jako v případě specifikací, čísla klonů odpovídají číslům, jak jsou uvedena na oficiálním webu[2] projektu společně s vysvětlením jednolivých chyb varianty. Pro přehlednost jsme bezchybnou variantu označili číslem 00.

[1] Seznam dostupný na adrese: `https://projects.kiv.zcu.cz/tbuis/web/page/uis#use-cases`

[2] Seznam poruchových klonů společně s možností stažení: `https://projects.kiv.zcu.cz/tbuis/web/page/download`

Specifikace	Popis	Porucha na klonech
1	*Přihlášení do aplikace* Student i učitel se přihlásí do aplikace přihlašovacími údaji, dostupnými v databázi systému. Dále se zkontroluje, zda systém pro účet s neexistujícím uživatelským jménem nebo neplatným heslem vypíše chybovou hlášku.	02
4	*Odepsání předmětu* Student se přihlásí, odepíše předmět v patřičné sekci systému. Předmět by následně měl zmizet v ostatních sekcích a to jak v pohledu studenta tak učitele.	04, 19, 25, 26, 28
6	*Zapsání předmětu* Student se přihlásí, zapíše předmět v patřičné sekci, který se následně zobarazí i v ostatních částech systému. Z učitelského pohledu by se student měl zobrazit na seznamu studentů daného předmětu.	25, 26, 28
8	*Registrace na zkoušku* Student se přihlásí do systémů a zapíše se na jeden z možných zkouškových termínů. Tento termín by se měl přesunout mezi již zapsané termíny. V učitelském pohledu bude student na seznamu zapsaných na konkrétní zkoušku a také by mělo být možno studenta ohodnotit.	22, 25, 26, 28
9	*Zobrazení spolužáků u zkoušky* Student se přihlásí a u zkoušky si může rozkliknout seznam všech účastníků.	
10	*Zrušení předmětu* Učitel po přihlášení klikne na tlačítko *Remove* u předmětu, od jehož výuky se chce odhlásit. Ten se přestane zobrazovat ve všech ostatních sekcích systému z jeho pohledu, až na jeho znovuzapsání. Student by u předmětu neměl najít jméno daného učitele, který se odhlásil.	26, 28
11	*Zobrazení studentů u předmětu* Učitel se přihlásí do systému a u předmětu je schopen si zobrazit seznam zapsaných studentů.	26, 28

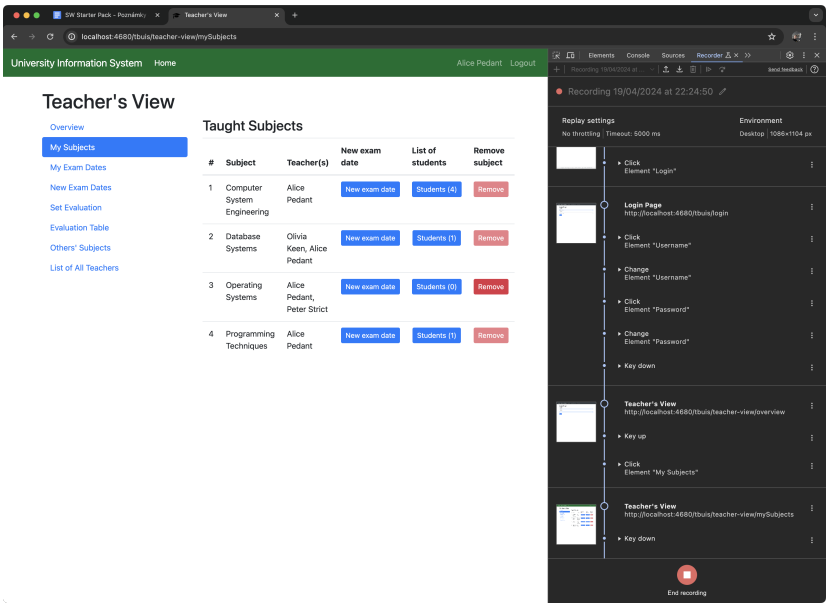
Tabulka 3.1: Specifikace pro generované testy - část 1

Specifikace	Popis	Porucha na klonech
12	*Zrušení zkoušky* Učitel se přihlásí a v sekci jeho přidělených zkoušek odstraní konkrétní termín. Tento termín by nyní neměl být vidět jak v učitelském tak studentském pohledu do systému.	20, 21, 23, 26, 28
17	*Přihlášení se k výuce předmětu* Učitel se přihlásí do systému a v seznamu předmětů se přihlásí k výuce daného předmětu. Ten by se poté měl zobrazit ve zbytku systému v rámci patřičných sekcí. Zároveň studenti by nyní měli u předmětu vidět jméno tohoto vyučujícího.	18, 24, 25, 26, 27, 28
18	*Zobrazení seznamu učitelů a předmětů, které vyučují* Přihlášený učitel je schopen si zobrazit seznam všeech učitelů.	27, 28

Tabulka 3.2: Specifikace pro generované testy - část 2

Číslo klonu	Porucha
00	Bez defektu
02	Překlep v nadpisu
04	Návrat na špatnou stránkau
18	Chybějící sloupec v tabulce
19	Náhodně chybějící tlačítko
20	Nefunkční tlačítko
21	Změna se nepropíše do UI
22	Nefunkční tlačítko
23	Smazání se nepropíše do DB
24	Přidání se nepropíše do DB
25	Tabulka studentů prázdná
26	Tabulka učitelů prázdná
27	Nesprávný výběr z DB
28	Mix chyb (včetně interní chyby systému)

Tabulka 3.3: Seznam poruchových klonů využitých pro testování.



Obrázek 3.1: Nahrávání scénáře za pomocí nástroje v Google Chrome

3.2 Nahrávání scénářů

Dle *specifikace* z předchozího bodu, vytvoří uživatel LLM generačního nástroje nahrávku jednotlivých kroků, které má test provést. V případě, že specifikace udává více uživatelských pohledů (*student / učitel*), nahraje uživatel tyto pohledy jednotlivě. Současný projekt pracuje s nahrávacím nástrojem v rámci vývojářských nástrojů webového prohlížeče *Google Chrome* (pro vývoj byla využita verze 124). Tento nástroj je stále experimentální funkce, takže nelze vyloučit možnost, že v následujících verzích již nemusí být dostupný. Díky této funkci lze nahrávat uživatelské vstupy a interakce s jednotlivými prvky v rámci webových stránek. Mimo jiné také umožňuje přidávat *asserty*, avšak tyto možnosti jsou nedostatečné a tedy v rámci této práce se omezíme pouze na údaje o interakci s prvky. Nahrávání zobrazeno na obr. 3.1. Nástroj je schopen vyexportovat výstup nahrávání v řadě formátů. Zde jsme zvolily JSON formát, který popisuje jednotlivé akce dle objektů. Nahrané scénáře uživatel vhodně pojmenuje a uloží do složky `input` programu, který je součástí tohoto projektu. Vytvořený program pro generování testů za pomocí LLM se stará o veškerou orchestraci generování i spouštění testů a také vytovření *šablon* pro testy, ze kterých se generují. Součástí těchto *šablon* je právě i uživatelova nahrávka. Ukázkovou strukturu `input` složky lze vidět ve výpisu 3.1, kde se nachází *šablona* pro *specifikace* 1 a 4. Význam *šablon* je diskutován v následující sekci.

Výpis 3.1: Ukázková struktura input složky

```
milan:dp\program\input$ ls -l
milan     448 Apr 21 20:52 ..
milan    3815 Mar 27 20:24 rec-spec-1-student.json
milan    3820 Mar 27 20:28 rec-spec-1-teacher.json
milan    6043 Mar 30 19:42 recording-spec-4-student.json
milan   10635 Mar 31 09:18 recording-spec-4-teacher.json
milan    1370 Apr 21 20:52 spec-1.txt
milan    1328 Mar 31 09:28 spec-4.txt

```

3.3 Výběr požadavků

3.3.1 Vytvoření nového testu

Primárním programem celé práce je soubor `test.py`, který se nalézá v kořenové složce programu (`/program`) a má konrétně 3 pracovní režimy:

1. Vyvtvoření šablony pro test
2. Vygenerování testu dle šablony
3. Spuštění sady testů

Právě první režim je v současném kroce důležitý. Jednotlivé režimy programu jsou spuštěny za pomocí argumentů (argumenty jednotlivých režimů ukázané ve výpisu 3.2). V případě *vytvoření* nového testu se za argument přidává název šalony testu. Tento název musí být unikání. Po potvrzení příkazu se ve vstupní složce vytvoří nová šablona pro test, kteoru uživatel může upravit. Celý příkaz je ukázaný ve výpisu 3.3. Vytvořenou šablonu uživatel zobrazí a upraví v *textovém editoru*.

Výpis 3.2: Hlavní režimy programu

```
milan:dp\program\input$ python3 test.py -n   #Nový test
milan:dp\program\input$ python3 test.py -i   #Generování
milan:dp\program\input$ python3 test.py -r   #Spuštění testů

```

Výpis 3.3: Vytvoření nového testu (šablony)

```
milan:dp\program\input$ python3 test.py -n specficication-1

```

Zdrojový kód 3.4: Vzor pro vyplnění šablony testu

```
Write Robot Framework scanario. Open page like in this JSON
    recording and then when you execute all the steps in the
    recording, do this:

- //TODO

{% include 'recording.json' %}

```

Do šablony je vložen základní prompt společně s požadavky, které uživatel může vyplnit (viz ukázka 3.4). Pod požadavky je defaultně vložena nahrávka `recording.json`. Tuto hodnotu nahradí uživatel za název souboru vložené nahrávky. Formát šablony jakožto vstupu pro prompt testu byl zvolen pro jednodušší parametrizaci. Šablona využívá jazyk *Jinja2*. V rámci vytvořených šablon v této práci byly tyto parametrizační vlastnosti využity například pro vytvoření pohledu *studenta* a *učitele*. Místo, které je v šabloně označeno jako `\\TODO` slouží pro napsání požadavků testu. Od uživatele se čeká, že tyto požadavky vypíše v odrážkách, čemuž odpovídá i formát *promptu*.

3.3.2 Vyplnění požadavků

3.4 Dotazování LLM

3.4.1 Prostředí pro spuštění

3.4.2 Dotazy

Spuštění testů 4

4.1 Spuštění testovacího programu a jeho orchestrace

Vyhodnocení výsledků

5

Budoucí vylepšení 6

Závěr 7

Bibliografie

[Cam+18] CAMPOS, José et al. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*. 2018, roč. 104, s. 207–235. ISSN 0950-5849. Dostupné z DOI: https://doi.org/10.1016/j.infsof.2018.08.010.

[EC06] ENGLER, Dawson; CHEN, David Yu. EXE: Automatically Generating Inputs of Death. In: *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*. New York, NY, USA: ACM, 2006, s. 322–335. Dostupné z DOI: 10.1145/1180405.1180445.

[Kat23] KATRIN KAHUR, Jennifer Su. *Java Unit Testing with AI: An AI-Driven Prototype for Unit Test Generation*. 2023. KTH Royal Institute of Technology.

[Li+23] LI, Raymond; ALLAL, Loubna Ben; ZI, Yangtian; MUENNIGHOFF, Niklas; KOCETKOV, Denis et al. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161*. 2023.

[LKF21] LUKASCZYK, Stephan; KROISS, Florian; FRASER, Gordon. An Empirical Study of Automated Unit Test Generation for Python. *CoRR*. 2021, roč. abs/2111.05003. Dostupné z arXiv: 2111.05003.

[Mat18] MATYÁŠ, Jiří. *Aplikace s možností injekce chyb pro ověřování kvality testů*. 2018. Dostupné také z: https://projects.kiv.zcu.cz/tbuis/web/files/uis/kp/DP-Matyas.pdf. Dipl. pr. University of West Bohemia. Accessed: 2024-04-18.

[Mic23a] MICROSOFT. *Dynamic symbolic execution - Visual Studio (Windows)*. 2023-03. Dostupné také z: %5Curl%7Bhttps://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/input-generation?view=vs-2022%7D. Čteno: 2023-10-10.

[Mic23b] MICROSOFT. *Test generation - Visual Studio (Windows)* [https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/test-generation?view=vs-2022]. 2023-03. [cit. 2023-10-10]. Dostupné z: https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/test-generation?view=vs-2022. Čteno: 2023-10-10.

[Pac+07] PACHECO, Carlos; LAHIRI, Shuvendu K.; ERNST, Michael D.; BALL, Thomas. Feedback-Directed Random Test Generation. In: *ICSE '07*. Washington, DC, USA: IEEE Computer Society, 2007. Dostupné také z: https://homes.cs.washington.edu/~mernst/pubs/feedback-testgen-icse2007.pdf.

[Par] PARÍZEK, Pavel. *Symbolic Execution* [Online]. [B.r.]. Dostupné také z: %5Curl%7Bhttps://d3s.mff.cuni.cz/legacy/teaching/program_analysis_verification/files/lecture03.pdf%7D. Lecture notes for Program Analysis and Verification course.

[Pou19] POUBOVÁ, Jitka. *Generování automatizovaných funkcionálních testů*. 2019. Dostupné také z: https://projects.kiv.zcu.cz/tbuis/web/files/uis/kp/BP-Poubova.pdf. University of West Bohemia. Accessed: 2024-04-18.

[Pro23] PROJECT, BigCode. *CodeML OpenRAIL-M 0.1 License*. 2023. Dostupné také z: %5Curl%7Bhttps://www.bigcode-project.org/docs/pages/model-license/%7D. Čteno: 23.10.2023.

[Sae18] SAES, Laurence. *Unit test Generation Using Machine Learning*. 2018. Dipl. pr. Universiteit van Amsterdam. https://research.infosupport.com/wp-content/uploads/Unit-test-generation-using-machine-Master-Thesis-Laurence-Saes.pdf.

[SMA05] SEN, Koushik; MARINOV, Darko; AGHA, Gul. CUTE: A Concolic Unit Testing Engine for C. In: *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2005. Dostupné také z: https://web.archive.org/web/20100629114645/http://srl.cs.berkeley.edu/~ksen/papers/C159-sen.pdf. Available at the University of Illinois at Urbana-Champaign.

[Sch+23] SCHÄFER, Max; NADI, Sarah; EGHBALI, Aryaz; TIP, Frank. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *arXiv preprint arXiv:2302.06527*. 2023. Dostupné také z: https://arxiv.org/pdf/2302.06527.pdf.

[Sid+23] SIDDIQ, Mohammed Latif et al. Exploring the Effectiveness of Large Language Models in Generating Unit Tests. *arXiv preprint arXiv:2305.00418*. 2023.

[SGA07] SUTTON, Michael; GREENE, Adam; AMINI, Pedram. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007. ISBN 0321446119.

[Šma19] ŠMAUS, Jakub. *Rozhraní pro administraci aplikace s možností injekce chyb*. 2019. Dostupné také z: `https://projects.kiv.zcu.cz/tbuis/web/files/uis/kp/DP-Smaus.pdf`. Dipl. pr. University of West Bohemia. Accessed: 2024-04-18.

[21] *Test Smells* [`https://testsmells.org/pages/testsmells.html`]. 2021. Dostupné také z: `https://testsmells.org/pages/testsmells.html`. Čteno: 23.10.2023.

[Vai20] VAIS, Radek. *Akceptační testování v projektu TbUIS*. 2020. Dostupné také z: `https://projects.kiv.zcu.cz/tbuis/web/files/uis/kp/DP-Vais.pdf`. Dipl. pr. University of West Bohemia. Accessed: 2024-04-18.

[Zho+06] ZHOU, Feng et al. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. Seattle, WA: USENIX Association, 2006, s. 45–60. Dostupné také z: `https://web.archive.org/web/20080829223442/http://cm.bell-labs.com/who/god/public_psfiles/pldi2005.pdf`. Available at Bell Labs.

Seznam obrázků

Seznam tabulek

Seznam výpisů