Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky

DIPLOMOVÁ PRÁCE

PLZEŇ, 2024 **Milan Horínek**

PROHLÁŠENÍ

Předkládám tímto k posouzení a obhajobě bakalářskou práci zpracovanou na závěr studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím odborné literatury a pramenů, jejichž úplný seznam je její součástí.

V Plzni dne 12. února 2024

. . . . . .
Milan Horínek

PODĚKOVÁNÍ

TDB

ANOTACE

TDB

Klíčová slova: jednotkové testy, LLM, strojové učení, automatizace

ABSTRACT

TBD

Key words: unit test, LLM, machine learning, automation

ZADÁNÍ

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. Navrhněte a implementujte automatizovaný nástroj využívající popis testu v přirozené řeči, LLM a případně další informace, který vygeneruje sadu jednotkových testů.
4. Ověřte kvalitu automaticky vytvořených testů zejména s ohledem na přesnost a úplnost.
5. Zhodnoťte možnosti současných LLM pro generování testů.

ASSIGNMENT

1. Familiarize yourself with existing Large Language Models (LLMs), their technology, and application possibilities.
2. Study the existing literature regarding the generation of unit tests, especially with regards to the use of neural networks and LLMs.
3. Familiarize yourself with existing sample programs with the possibility of error injection and select a suitable test program. Design and implement an automated tool using the test description in natural language, LLMs, and possibly additional information, which will generate a set of unit tests.
4. Verify the quality of the automatically created tests, particularly with regards to accuracy and completeness.
5. Evaluate the possibilities of current LLMs for test generation.

Obsah

Zkratky

PUT Parameterized Unit Test. 7

Slovník pojmů

filtr Ve vývoji softwaru a testování se odkazuje na mechanismy nebo kritéria používaná k selektivnímu výběru testovacích vstupů nebo k rozhodování, které výstupy testů jsou relevantní pro další analýzu. Filtry mohou být použity k odstranění redundantních, nelegálních nebo jinak nežádoucích vstupů z procesu generování testů, což zvyšuje efektivitu testování tím, že se zaměřuje pouze na vstupy, které mohou odhalit chyby nebo porušení kontraktů. 8

kontrakt V kontextu jednotkových testů odkazuje na formálně definované podmínky nebo pravidla, které musí být dodrženy během vykonávání kódu. Kontrakty mohou specifikovat, co funkce očekává od svých vstupů a jaké výstupy nebo stavy by měla produkce kódu způsobit. Použití kontraktů pomáhá zajistit, že kód se chová podle očekávání a usnadňuje identifikaci chyb, když tyto podmínky nejsou splněny. 8

LLM Velký jazykový model (LLM - large language model) je typ modelu strojového učení, který je navržen tak, aby rozuměl a generoval text v přirozeném jazyce. Tyto modely jsou trénovány na obrovských datasetech a jsou schopny provádět různé úkoly, jako je textová klasifikace, generování textu, strojový překlad a další. 8, 9

testové pachy Unit Test Smells jsou špatné návyky nebo postupy, které se mohou objevit při psaní jednotkových testů. Tyto ”smelly” často vedou k neefektivním nebo nespolehlivým testům a mohou ztížit údržbu testovacího kódu. Příklady zahrnují Duplicated Asserts, Empty Tests a General Fixture. 9

1 Úvod

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

2 Provedená práce v problematice

2.1 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 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. [1]

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. [2] 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. [3] [4]

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. [5] [6] [7]

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. [8]

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. [9] [10]

2.1.2 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“[11], 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.

2.1.3 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.

2.2 Vydané publikace

Jeden z poměrně nedávno vydaných článků (září 2023) nazvaný „An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation“ [12] 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 [13]. 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!“ [13] 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“ [14] 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 vygenerovaných testech často objevovaly tzv. testové pachy, jako jsou *duplicitní tvrzení* a *prázdné testy* [15].

2.2.1 Srovnání výsledků

Výsledky diskutovaných studií v předchozím bodě jsme srovnali v tabulce 2. 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 [12] 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. [16] také využívá Javu a s modelem *gpt-3.5-turbo* dosahuje podobného pokrytí kódu a úspěšnosti jako *Codex* v práci [14].

2.3 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 3.

Jedním z často používaných modelů v předchozích pracích je *StarCoder* a *StarCoderBase*, diskutovaný již v sekci 2. *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** [17] umožňuje ho využívat pro množšství 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.

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 kterýcg je uvedeno, že model nelze použít k vytvoření služby, která by konkurovala vlastním službám společnosti Meta.

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.

2.3.1 Srovnání modelů

Z diskutovaných modelů jsme 3 z nich (*CodeLlama, StarCoderBase a GPT-4*) podrobili vlastnímu testu, kdy jako testovací sada byli využity 4 JavaScript funkce a to ve 3 různých verzích. V první verzi se jednalo o *white box* skript, tedy že modelu byl poskytnut celý obsah funkcí. Dále byli modelům dány pouze *specifikace* funkcí bez jejich těla a poslední verzí je *white box kód s vloženými chybami*. Správná a chybová verze kódu také byla využita pro zhodnocení vygenerovaných funkcí.

Model	**CodeLlama**		**GPT-4**		**StarCoderBase**	
Kód	Správný	Chybový	Správný	Chybový	Správný	Chybový
Specifikace	4/4	3/4	4/4	3/4	3/4	2/4
White box	4/4	3/4	4/4	3/4	3/4	3/4
Err white box	4/4	3/4	4/4	3/4	3/4	3/4

Tabulka 1: Srovnání výsledků generování jednotkových testů za pomocí vybraných modelů.

Výsledky těchto testů lze nalézt v tabulce 1. Zatímco u správného kódu bylo hodnoceno, kolik funkcí prošlo testy, tak u funkcí s vloženými chybami se počítá, kolik z nich testy neprošlo. Model *CodeLlama* byl schopný odhalit chybu ve všech funkcích až na jednu. Všechny správné funkce prošli. Je zde však nutné dodat, že generování je velmi pomalé a výstup velice jednoduchý. Nebyl zde náznak o netriviální assert, ovšem pokud se model bude správně promptovat, lze jeho vygenerovaný test dobře využít. *GPT-4* poté měl stejnou úspěšnost jako předchozí model, ovšem i zde byl problém s netriviální asercí. Jednotlivé errory jsou však dobře otestované a výhodou je, že model je rychlý a není tedy v případě nevhodně vygenrovaného testu ho ziterovat a vylepšit na bázi zpětné vazby (*multi-shot*). Při tomto přístupu je poté schopen i netriviálních asercí. Je však potřeba tento model dobře promptovat a výstup může být složitější pro parsování. Model *StarCoderBase* dle výsledků má nejméně úspěšných testů. Ovšem když zanalyzujeme výstupní kód (testy), dojdeme k zajímavému zjištění. Tento model dokázal odchytit i netriviální chyby přímo ze specifikace a zároveň se mu povedlo obejít nepřesnosti ve specifikaci, tedy neúspěšné testy na správném kódu byli stále v rámci specifikace, jen ne ve smyslu, jak byla speficikace zamýšlena. Subjetivně řečeno, *StarCoderBase* dělá správné úsudky a i kód je velice obsáhlý. S přihlédnutím na jeho otevřenost a licenci se jeví jako vhodný model právě pro naši práci.

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: 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 3: Přehled a srovnání modelů generujících kód

Reference

[1] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[2] P. Parízek, "Symbolic execution." Online. Lecture notes for Program Analysis and Verification course.

[3] Microsoft, "Dynamic symbolic execution - visual studio (windows)," March 2023. Čteno: 2023-10-10.

[4] Microsoft, "Test generation - visual studio (windows)." `https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/test-generation?view=vs-2022`, 3 2023. Čteno: 2023-10-10.

[5] D. Engler and D. Y. Chen, "Exe: Automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, (New York, NY, USA), pp. 322–335, ACM, 2006.

[6] K. Sen, D. Marinov, and G. Agha, "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. Available at the University of Illinois at Urbana-Champaign.

[7] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, "Safedrive: Safe and recoverable extensions using language-based techniques," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, (Seattle, WA), pp. 45–60, USENIX Association, 2006. Available at Bell Labs.

[8] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE '07*, (Washington, DC, USA), IEEE Computer Society, 2007.

[9] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Information and Software Technology*, vol. 104, pp. 207–235, 2018.

[10] S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of automated unit test generation for python," *CoRR*, vol. abs/2111.05003, 2021.

[11] L. Saes, "Unit test generation using machine learning," Master's thesis, Universiteit van Amsterdam, 2018. `https://research.infosupport.com/wp-content/uploads/Unit-test-generation-using-machine-Master-Thesis-Laurence-Saes.pdf`.

[12] M. Schafer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *arXiv preprint arXiv:2302.06527*, 2023.

[13] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, *et al.*, "Starcoder: may the source be with you!," *arXiv preprint arXiv:2305.06161*, 2023.

[14] M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, "Exploring the effectiveness of large language models in generating unit tests," *arXiv preprint arXiv:2305.00418*, 2023.

[15] "Test smells." `https://testsmells.org/pages/testsmells.html`, 2021. Čteno: 23.10.2023.

[16] J. S. Katrin Kahur, "Java unit testing with ai: An ai-driven prototype for unit test generation," 2023.

[17] B. Project, "Codeml openrail-m 0.1 license," 2023. Čteno: 23.10.2023.