Issue 18, part 1, 2025

https://doi.org/10.23939/sisn2025.18.229

УДК 004.415:004.89

ПОРІВНЯЛЬНИЙ АНАЛІЗ ГЕНЕРАЦІЇ СМАРТ-КОНТРАКТІВ SOLIDITY ЗА ДОПОМОГОЮ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ НА ОСНОВІ ФОРМАЛЬНИХ АЛГЕБРАЇЧНИХ СПЕЦИФІКАЦІЙ

Володимир Песчаненко [1], Ольга Коннова[2]

[1, 2] Херсонський державний університет,
кафедра комп'ютерних наук та програмної інженерії, Херсон, Україна
[1] E-mail: vladim@ksu.ks.ua, ORCID: 0000-0003-1013-9877
[2] E-mail: okonnova@ksu.ks.ua, ORCID: 0000-0001-5590-9527

Дана стаття присвячена порівняльному аналізу автоматичної генерації смарт-контрактів на мові Solidity за допомогою великих мовних моделей на основі двох підходів: текстових описів природною мовою та формальних алгебраїчних специфікацій. У ході роботи було проаналізовано смарт-контракти, згенеровані великими мовними моделями (ChatGPT-4, Claude 3.7 Sonnet, DeepSeek-V3), а також інструментом на основі штучного інтелекту GitHub Copilot, з оцінкою їхньої синтаксичної коректності та відповідності початковим вимогам. Безпека смарт-контрактів оцінювалася за допомогою статичного аналізу з використанням інструменту Slither, для виявлення потенційних вразливостей, зокрема атак повторного входу та арифметичних переповнень.

Для створення формальної моделі ми використовували інсерційне моделювання, яке дозволяє формально описувати поведінку смарт-контрактів через алгебраїчні специфікації. Для покращення інтерпретації великими мовними моделями алгебраїчних специфікацій, нами було запропоновано підхід промптингу з невеликою кількістю прикладів, який передбачає надання LLM демонстраційних прикладів алгебраїчних специфікацій і відповідного коду Solidity для покращення інтерпретації формальних моделей. Запропонований підхід дозволив підвищити точність трансляції формальних специфікацій у код, зменшити кількість синтаксичних помилок та забезпечити кращу відповідність до вихідної моделі.

У ході дослідження було виявлено, що текстові запити є простішими для швидкої генерації, однак менш надійними. Через властиву їм двозначність вони частіше призводять до появи прихованих вразливостей. Натомість формальні моделі вимагають більше зусиль для налаштування великих мовних моделей та усунення помилок інтерпретації, проте дозволяють значно точніше відобразити логіку смарт-контракту, що забезпечує більш якісну генерацію контрактів зі складною поведінкою.

Ключові слова – блокчейн, смарт-контракти, штучний інтелект, великі мовні моделі, алгебраїчні специфікації, інсерційне моделювання, генерація смарт-контрактів, Solidity

Постановка проблеми

Смарт-контракти є основою децентралізованих технологій, забезпечуючи автоматизацію та безпечне виконання угод у блокчейн-середовищі. Однак процес розробки смарт-контрактів є досить складним та трудомістким завданням. Він вимагає від розробників наявності знань специфічних мов програмування, які використовуються для створення смарт-контрактів для різних блокчейн платформ.

Наприклад, Solidity є стандартом для Ethereum, PyTeal використовується в Algorand, а Go та JavaScript для Hyperledger Fabric. Крім того, кожна з цих платформ має власні фреймворки для створення смарт-контрактів, що ще більше ускладнює крос-платформне розгортання. Адже окрім відмінностей у синтаксисі, необхідно враховувати специфічні інструменти, середовища розробки та архітектурні підходи кожної платформи. Ще одним важливим аспектом у розробці смарт-контрактів є забезпечення високого рівня безпеки. Смарт-контракти часто управляють значними фінансовими активами, тому будь-яка вразливість у їхньому коді може призвести до серйозних наслідків, таких як втрата коштів.

Автоматизована генерація коду смарт-контрактів є перспективним рішенням цих проблем. Це процес, який використовує інструменти, алгоритми та фреймворки для автоматичної генерації коду смарт-контракту з високорівневої специфікації або опису природною мовою в код для блокчейн-платформ. Вона дозволяє мінімізувати або виключити необхідність ручного програмування, тим самим зменшуючи ризики людських помилок і прискорюючи процес розробки. Одним з підходів до автоматичної генерації коду смарт-контактів є використання штучного інтелекту. Використання генеративного ШІ для створення програмного коду набуває все більшої популярності, оскільки сучасні великі мовні моделі (LLM), такі як GPT-4, OpenAI Codex, Code Llama та інші, можуть генерувати код на основі запитів, сформульованих природною мовою. Цей підхід використовує моделі штучного інтелекту для інтерпретації введених користувачем даних, аналізу вимог і створення коду. Також, LLM можуть аналізувати велику кількість прикладів існуючих смарт-контрактів, виявляти закономірності й пропонувати користувачу окремі функції або повноцінні смарт-контракти на основі текстового опису. Використання ШІ може значно прискорити створення надійного коду, зменшити кількість помилок та вразливостей.

Основною проблемою автоматичної генерації смарт-контрактів є забезпечення їхньої коректності та відповідності бізнес-логіці. Згенеровані контракти можуть містити логічні помилки, вразливості або не відповідати початковому задуму, що робить критично важливим питання якості та точності генерації. Вхідні дані, які використовуються для генерації коду, відіграють ключову роль у кінцевому результаті. Структура та якість запиту користувача до LLM суттєво впливає на результат генерації смарт-контракту.

Аналіз останніх досліджень та публікацій

Автоматизація розробки програмного забезпечення за допомогою ШІ досягла швидкого прогресу, особливо з моделями, здатними генерувати код з текстових описів. Серед ключових напрямів застосування LLM у програмуванні виділяються генерація коду, автодоповнення, переклад коду, редагування, узагальнення, а також виявлення дефектів (Wong та ін., 2023). Генеративний ШІ може скоротити час початкової розробки коду до 45 % за рахунок автоматизованої генерації коду та покращити його якість на 65 % (Modi, 2024). У контексті блокчейн-технологій ці можливості дозволяють автоматизувати створення коду для смарт-контрактів.

Chatterjee та Ramamurthy (2024) у своїй роботі досліджують застосування великих мовних моделей (LLM) для створення смарт-контрактів Solidity на блокчейні Ethereum. Дослідження спрямоване на оцінку точності, надійності та безпеки згенерованих смарт-контрактів. Автори протестували кілька LLM, включаючи GPT-3.5, GPT-4, GPT-4-o, Cohere, Mistral, Gemini і Claude. У якості даних для генерації коди використовувалися описові запити у вигляді текстових інструкцій, сформульованих природною мовою, та структуровані запити, що подаються у вигляді псевдокоду. У ході роботи автори дійшли висновку, що великі мовні моделі мають значний потенціал для автоматичної генерації смарт-контрактів, але наразі існують певні обмеження, які потрібно враховувати. Одним з основних є те, що більшість моделей не враховували питання безпеки, якщо це не було прямо вказано в запиті. Дослідження також підкреслює, що хоча LLM можуть ефективно виконувати прості завдання, їм важко справлятися з більш складними вимогами до кодування, а якість згенерованого коду може бути нестабільною. Fadi та ін. (2024) у ході свого дослідження можливостей використання GPT-4 для перекладу юридичного договору в код смарт-контракту також зауважили, що згенерований код не відповідає стандартам для використання в реальних умовах через наявність логічних помилок та недоліків у функціональності.

Kang та ін. (2024) у своїй роботі описують можливості використання великих мовних моделей для виділення логіки прийняття рішень і генерації коду смарт-контрактів із тексту страхових полісів у галузі охорони здоров'я. Автори використовують LLM для аналізу тексту документів й створення стислого викладення змісту поліса; для виділення високорівневої логіки з тексту документів, використовуючи різні формалізми, такі як Notation3 (N3) та Clinical Quality Language (CQL), та для генерації коду смарт-контрактів на мові Solidity і юніт-тестів для перевірки правильності коду. Автори використовували GPT-3.5 Turbo, GPT-3.5 Turbo 16K, GPT-4, GPT-4 Turbo та Code LlaMA. У результаті дослідження автори також відзначають, що великі мовні моделі добре справляються з більш простими завданнями, як генерація стислого викладу тексту, але стикаються з труднощами при роботі зі складною логікою та великими обсягами тексту. Моделі іноді пропускають важливі умови, неправильно реалізують логіку або генерують елементи, які не згадуються в документі.

Отже, проаналізувавши дослідження щодо використання LLM для генерації коду смарт-контрактів, можна відзначити, що основною проблемою є неточність результатів, особливо для контрактів зі складною логікою, їх невідповідність специфікаціям та наявність вразливостей у безпеці.

У цій роботі ми розглядаємо використання формальних моделей для генерації смарт-контрактів. На відміну від традиційних текстових запитів природною мовою, формалізовані моделі мають чітку структуру та семантику, що знижує неоднозначність і дозволяє точно передати бізнес-логіку смарт-контракту. Вони дозволяють структуровано описати поведінку смарт-контракту, що потенційно дозволяє зменшити кількість помилок при автоматичному створенні програмного коду. Однак, важливим аспектом в процесі генерації коду на основі формальних моделей є коректна інтерпретація великими мовними моделями формальних синтаксичних конструкцій.

Формулювання цілі статті

Метою цієї роботи є експериментальне порівняння автоматичної генерації смарт-контрактів за допомогою LLM на основі двох підходів: текстових описів природною мовою та формальних алгебраїчних специфікацій.

Для досягнення цієї мети, ми поставили перед собою такі завдання:

1. Дослідити сучасні підходи до генерації смарт-контрактів мовою Solidity за допомогою LLM, зокрема на основі текстових описів і формальних специфікацій.
2. Підготувати вхідні дані для генерації смарт-контрактів (приклади запитів більш детально описані у наступному розділі).
3. Здійснити генерацію смарт-контрактів за допомогою обраних LLM на основі двох підходів.
4. Оцінити згенеровані контракти за метриками: синтаксична коректність, відповідність вимогам, відсутність вразливостей.
5. Виконати порівняльний аналіз результатів генерації та сформулювати висновки щодо ефективності використання LLM для генерації смарт-контрактів залежно від типу вхідного запиту, а також виявити переваги та обмеження кожного з підходів.

Ми очікуємо, що формальні моделі забезпечать більш точну та безпечну генерацію смарт-контрактів порівняно з текстовими описами, оскільки вони зменшують ризик логічних помилок та двозначного трактування вхідних даних.

Виклад основного матеріалу

Дослідження зосереджено на аналізі генерації смарт-контрактів мовою Solidity за допомогою великих мовних моделей шляхом порівняння двох підходів: використання природномовних описів вимог (текстових запитів) і формальних специфікацій. Методологія охоплює підготовку вхідних даних, вибір інструментів для генерації та оцінку згенерованих результатів за допомогою специфічних метрик для забезпечення якості та надійності.

Вхідні дані:

- Текстовий запит: пояснення бажаної функціональності смарт-контракту, написане природною мовою. Цей опис окреслював мету системи, ключові функції (наприклад, стейкінг, розподіл винагород, розблокування токенів) та примітки щодо реалізації (наприклад, відповідність стандарту ERC-20).
- Формальна модель смарт-контракту: формальна модель смарт-контракту, створена з використанням інсерційного моделювання (IMS) (детально описана у наступному розділі).

Для генерації смарт-контрактів на Solidity було використано великі мовні моделі: ChatGPT-4, Claude 3.7 Sonnet, DeepSeek-V3, а також інструмент на основі штучного інтелекту GitHub Copilot. Усі моделі отримали однакові вхідні дані (текстовий опис і формальну модель) для генерації смарт-контрактів на Solidity. Варто зазначити, що GitHub Copilot не є самостійною великою мовною моделлю, а інструментом на основі штучного інтелекту, який інтегрує можливості LLM у середовище програмування. Його результати наведено для порівняння з безпосереднім використанням LLM у задачі генерації смарт-контрактів.

Для оцінки ефективності кожної LLM ми згенерували по 5 смарт-контрактів для кожної конфігурації (модель × тип запиту) – як на основі текстових описів вимог, так і з використанням алгебраїчних специфікацій. Згенеровані смарт-контракти оцінювалися за трьома ключовими метриками для забезпечення якості та придатності до розгортання:

- Синтаксична коректність: згенерований код Solidity не повинен містити синтаксичних помилок і повинен бути успішно скомпільований за допомогою стандартного компілятора Solidity.
- Логічна відповідність вимогам: згенерований контракт має точно реалізовувати функціональність, описану в текстових вимогах і формальній моделі, включаючи всі зазначені дії та властивості.
- Відсутність вразливостей: згенерований смарт-контракт не повинен містити вразливостей, таких як повторний вхід, переповнення/недоповнення цілих чисел, неналежний контроль доступу тощо. Ми використовували інструмент статичного аналізу коду смарт-контрактів Slither для перевірки згенерованого коду на наявність вразливостей. Кількість виявлених Slither вразливостей усереднювалася для п’яти контрактів у межах кожної конфігурації. Отримані показники відображають загальний рівень безпеки згенерованого коду та дають змогу порівняти надійність різних підходів.

Експеримент включав наступні етапи:

1. Підготовка запиту передбачає створення повних та чітко структурованих запитів, які мали забезпечити максимально точну генерацію коду відповідно до заданих вимог.

2. Генерація коду полягає у тому, що кожній LLM надавалися однакові запити для генерації коду.

3. Оцінка полягала у обранні контрактів, що оцінювалися за синтаксичною коректністю, логічною відповідністю та безпекою за допомогою описаних метрик та інструментів. Узагальнені результати наведені в таблиці 1.

Генерація на основі текстового запиту

Для генерації смарт-контракту на основі запиту природньою мовою, було використано детальний опис бажаної поведінки смарт-контракту. Запит враховує усі умови, які мають бути виконані у контракті. Включає типові елементи токеноміки: стейкінг, винагороди, поступове розблокування, а також додано захист від зловживань і сумісність із ERC-20 – стандарт для створення та випуску токенів у блокчейні Ethereum.

Такий підхід належить до інструктивного промптингу (instruction-based prompting), де запит задається у вигляді повного опису очікуваної логіки поведінки системи, і використовується як специфікація до генерації коду. Цей стиль передбачає одноетапну генерацію коду на основі детального текстового опису вимог без використання прикладів чи ланцюгів запитів.

Приклад запиту до LLM для генерації смарт-контракту:

"Згенеруй смарт-контракт на Solidity (версія 0.8.20) для стейкінгу ERC-20 токенів із підтримкою нарахування винагород. Контракт повинен мати наступний функціонал:

- *користувачі можуть стейкати токени ERC-20, якщо мають достатню кількість токенів на гаманці. Мінімальна сума стейкінгу дорівнює 100 токенів.*
- *користувачам нараховується винагорода у розмірі 1 % на день від суми їхніх токенів на стейкінгу.*
- *нарахування винагороди відбувається пропорційно фактичному часу блокування токенів. Тобто, якщо токени перебували у стейкінгу X днів, винагорода розраховується як (Сума Стейкінгу * 1 % * X днів);*
- *виведення токенів можливе лише після завершення періоду стейкінгу (30 днів з моменту стейкінгу), та за умови, що користувач має достатню кількість токенів на стейкінгу.*

Включи захист від повторного входу для всіх критичних операцій. Забезпеч мінімізацію витрат газу, належну обробку помилок та генерацію подій для всіх ключових операцій."

Ми генеруємо код контракту з використанням LLM: ChatGPT-4, Claude 3.7 Sonnet, DeepSeek-V3, та інструменту GitHub Copilot. Далі аналізуємо результати генерації на основі запиту, написаного природньою мовою відповідно до визначених метрик оцінки. Для перевірки коду на наявність вразливостей ми використовуємо інструмент статичного аналізу кода – Slither. Він дозволяє виявляє поширені ризики безпеки, такі як повторний вхід, цілочисельне переповнення тощо.

Для перевірки правильності розрахунку винагород контрактом відповідно до заданих вимог ми розгорнули згенеровані контракти, провели симуляцію та проаналізували результати. Таким чином було виявлено приклад некоректного розподілу винагород: у контракті, згенерованому GitHub Copilot, щоденна винагорода у розмірі 1 % була поділена на 365 днів, що призводить до значно меншого розміру винагороди (наприклад, для 1 000 токенів замість очікуваних 10 токенів на день нараховувалося лише близько 0,027 токенів). Це пояснюється тим, що LLM може неправильно інтерпретувати "1 % на день" як частину річної ставки (наприклад, 1 % на день протягом 365 днів дорівнює річній ставці, тому при розрахунку відбувається ділення на 365). Подібна помилка виникає через двозначність у формулюванні запиту та тенденцію LLM застосовувати поширені фінансові патерни, засвоєні з даних навчання.

Оцінка згенерованих смарт-контрактів:

- Синтаксична коректність: не всі смарт-контракти, згенеровані обраними LLM, є синтаксично коректними і успішно компілюються без помилок (були наявні певні неточності, наприклад, TypeError через відсутність аргументу для конструктора Ownable у коді контрактів, згенерованих за допомогою GitHub Copilot та Claude 3.7 Sonnet. У деяких згенерованих конрактах виникали помилки через некоректну роботу з подіями (events), структурні помилки (такі, як незакриті фігурні дужки). Однак, вони були усунуті після вказання LLM про наявність помилки). Усі контракти використовують сучасні конструкції мови та інтегрують стандартні бібліотеки OpenZeppelin.
- Логічна відповідність вимогам: згенеровані контракти реалізують більшість вимог (стейкінг, захист, ERC-20), але мають помилки в реалізації логіки розблокування токенів (так, контракти, згенеровані GitHub Copilot та ChatGPT-4, мали проблему з некоректним розрахунком та подвійним нарахуванням винагород), проблеми з ініціалізацією пула винагород. Були наявні певні неточності, наприклад, додавання в код локальних змінних, які не використовуються.
- Відсутність вразливостей: хоча згенеровані контракти використовували модифікатори nonReentrant та вбудовані перевірки, доступні починаючи з Solidity 0.8.0, для захисту, аналіз Slither виявив потенційні вразливості, зокрема пов'язані з повторним входом через порушення шаблону "Checks-Effects-Interactions", а також проблеми точності розрахунків винагород через "ділення перед множенням". Також були зафіксовані попередження про наявність "мертвого коду" у згенерованих контрактах.

```
INFO:Detectors:
Reentrancy in StakingContract.stake(uint256) (StakingContract.sol#42-66):
        External calls:
        - require(bool,string)(stakingToken.transferFrom(msg.sender,address(this),_amount),Token transfer failed) (StakingContract.sol#58)
        State variables written after the call(s):
        - stakes[msg.sender].amount += _amount (StakingContract.sol#61)
        StakingContract.stakes (StakingContract.sol#27) can be used in cross function reentrancies:
        - StakingContract.getPendingReward(address) (StakingContract.sol#98-109)
        - StakingContract.getStakeInfo(address) (StakingContract.sol#122-125)
        - StakingContract.stakes (StakingContract.sol#27)
        - stakes[msg.sender].timestamp = block.timestamp (StakingContract.sol#62)
        StakingContract.stakes (StakingContract.sol#27) can be used in cross function reentrancies:
        - StakingContract.getPendingReward(address) (StakingContract.sol#98-109)
        - StakingContract.getStakeInfo(address) (StakingContract.sol#122-125)
        - StakingContract.stakes (StakingContract.sol#27)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```

Рис. 1. Попередження Slither про вразливість до повторного входу у функції stake() смарт-контракту, згенерованого з використанням ChatGPT-4

Помилка повторного входу в згенерованому LLM контракті виникає через те, що модель, хоча й використовує захисний модифікатор nonReentrant, не дотримується архітектурного шаблону "Checks-Effects-Interactions", який передбачає структурування функцій контракту в певному порядку: Перевірки (валідація вхідних даних та умов), Ефекти (зміна стану контракту) та Взаємодії (виклик інших контрактів або передача активів) ("Checks Effects Interactions", 2025). У згенерованому контракті виконується зовнішній виклик до оновлення внутрішнього стану контракту.

Результати експерименту показали, що генерація смарт-контрактів на основі текстових запитів, сформульованих природною мовою, може забезпечити базову функціональність і компіляцію коду, проте не гарантує повну логічну відповідність специфікації та високий рівень безпеки без додаткової перевірки й доопрацювання. Усі моделі допускають помилки в реалізації складних логічних вимог і можуть не враховувати специфічні особливості Solidity.

Важливим аспектом генерації смарт-контрактів на основі текстового запиту є точність його формулювання. Оскільки контракти зазвичай реалізують складну логіку, будь-які неточності в описі можуть призвести до некоректної інтерпретації з боку великих мовних моделей. Варто зазначити, що текстові описи, написані природною мовою, можуть бути складними для обробки LLM через їхню неоднозначність, неповноту та залежність від контексту.

Наприкінці статті ми наводимо порівняння двох підходів відповідно до визначених критеріїв (Таблиці 1-2).

Генерація коду контракту з використанням формальної моделі

Далі ми використовуємо формальну модель смарт-контракту для генерації коду за допомогою LLM, щоб порівняти її ефективність із текстовим підходом, описаним у попередньому розділі.

Формальна модель — це математично строге представлення системи, яке описує її поведінку, стан і взаємодію компонентів через чітко визначені правила та специфікації, зазвичай із використанням математичних методів. У контексті смарт-контрактів модель дозволяє однозначно визначити функціональні вимоги, усуваючи двозначність, яка може виникати при використанні природної мови, і забезпечує основу для автоматизованої генерації коду з високою точністю.

Формальна модель, використана для генерації смарт-контракту, була побудована у вигляді алгебраїчної специфікації, що реалізує підхід інсерційного моделювання (IMS) (Letichevsky та ін., 2016). У попередніх дослідженням ми представляли можливості використання IMS для формальної верифікації смарт-контрактів, згенерованих за допомогою ШІ (Peschanenko та ін., 2025).

Система IMS базується на концепції взаємодії агентів та середовища. Агенти взаємодіють один з одним та з середовищем відповідно до правил, формалізованих у вигляді дій та алгебри поведінки. У контексті смарт-контрактів агенти представляють учасників системи (наприклад, користувачів, інвесторів тощо). Поведінкові рівняння, виражені в алгебрі поведінки, описують послідовність дій і переходи станів системи.

Агенти – це сутності в системі, які виконують дії. Кожен агент представляє учасника або компонент. Атрибути – це властивості агентів, які визначають їхній стан.

У нашій моделі смарт-контракту ми визначили таких агентів:

- Користувач, що представляє окремого користувача, який розміщує токени на стейкінг, заробляє винагороди та запитує розблокування токенів.
- Контракт, який представляє сам смарт-контракт, який управляє глобальним станом, таким як загальна кількість поставлених на стейкінг токенів та пул винагород.

Типи агентів у системі IMS представлені наступним чином:

```
User:obj(
 balance:real,        // Баланс токенів користувача
 stakedToken:real,    // Кількість токенів користувача на стейкінгу
 reward:real,         // Накопичені винагороди користувача
 lastRewardTime:int,  // Мітка часу останнього розрахунку винагороди
 stakeTime:int        // Мітка часу початку стейкінгу
)

Contract:obj(
      totalStaked:real,     // Загальна кількість токенів, розміщених на стейкінг
                            // від усіх користувачів
 totalRewardPool:real, // Кількість токенів для виплати винагород (пул токенів
                       // для виплати винагород)
 minStakeAmount:real,  // Мінімальна сума стейкінгу (100 токенів)
 dailyRewardRate:real, // Відсоток винагороди (1% в день)
 stakingPeriod: real,  // Період стейкінгу (30 днів)
 currentTime:int,      // Поточний час у секундах
      secondsPerDay:real    // Кількість секунд у добі
)
```

Кожен агент може виконувати певні дії, які змінюють його стан, наприклад змінюють значення атрибутів агента.

Дії описують операції, які можуть виконувати агенти, змінюючи свої атрибути. Кожна дія є трійкою Хоара $\alpha \rightarrow< P > \beta$, де α – передумова, що визначає стан, коли протокол може бути застосований; β – це постумова, яка визначає перехід системи до нового стану; P – це процес, що ілюструє цей перехід. α та β представлені логічними виразами базової мови та визначають умови на множині станів системи. Кожна дія визначає властивості системи і може розумітися як твердження темпоральної логіки: якщо передумова істинна, то процес дії може початися, а після його успішного завершення постумова має бути істинною (Letichevsky та ін., 2005).

Приклад дії unstake в алгебраїчній формі представлена наступним чином:

```
unstake(user, contract) = (
    (user.stakedToken > 0 && contract.currentTime >= user.stakeTime +
     contract.stakingPeriod)->
    <'Tokens unstake'>
    (contract.totalStaked = contract.totalStaked - user.stakedToken;
     contract.totalRewardPool = contract.totalRewardPool - user.reward;
     user.balance = user.balance + user.stakedToken + user.reward;
     user.stakedToken = 0;
     user.reward = 0
)
```

Опис дії unstake: користувач знімає токени, які були на стейкінгу та накопичені винагороди після завершення визначеного періоду.

- Передумова: кількість токенів на стейкінгу користувача має бути більше за 0, а поточний час має перевищувати час початку стейкінгу плюс період стейкінгу (30 днів). Тобто визначений період стейкінгу завершився.
- Процес <'Tokens unstake'>: ілюструє перехід системи від стану до виконання дії до стану після її виконання.

- Постумова: змінює стан агентів contract та user. Баланс користувача збільшується на суму застейкованих токенів і накопичених винагород, загальна кількість токенів на стейкінгу у контракті зменшується на суму застейкованих токенів, пул винагород зменшується на суму нарахованої винагороди, а кількість токенів на стейкінгу та накопичені винагороди користувача скидаються до 0.

Дія calculateReward:

```
calculateReward (user) = (
    (user.stakedToken > 0) ->
    <'Calculate user`s rewards for staking'>
    (user.reward = user.reward + ((user.stakedToken * contract.dailyRewardRate *
    (contract.currentTime - user.lastRewardTime)) / (100 *
     contract.secondsPerDay)),
    user.lastRewardTime = contract.currentTime
)
```

Опис дії calculateReward – оновлення накопичених винагород користувача на основі часу, протягом якого його токени перебувають на стейкінгу, з урахуванням щоденної ставки винагороди 1 %.

- Передумова – кількість токенів на стейкінгу користувача (user.stakedToken) має бути більше 0.
- Процес – ілюструє перехід системи від стану до виконання дії до стану після її виконання.
- Постумова – накопичені винагороди користувача (user.reward) збільшуються на суму, розраховану як 1 % від стейканих токенів (user.stakedToken) за кількість днів, що минули з моменту останнього оновлення (contract.currentTime – user.lastRewardTime). Мітка часу останнього оновлення винагороди (user.lastRewardTime) оновлюється до поточного часу (contract.currentTime).

Поведінка визначає послідовність виконання дій. У межах Insertion Modeling System (IMS) це формалізується за допомогою алгебри поведінок – двосортної універсальної алгебри. Алгебра має основні операції (Letichevsky та ін., 2016):

1. Префіксна композиція: a·u, де a — дія, u — поведінка після виконання a.
2. Недетермінований вибір поведінки: u+v, асоціативна, комутативна, ідемпотентна операція вибору, де u та v – поведінки .
3. Паралельна композиція: {u ‖ v}, моделює одночасне виконання u та v.
4. Послідовна композиція: u ; v, виконання u перед v.

Для реалізації шаблону "Checks-Effects-Interactions" (CEI) на прикладі функції Stake, ми описали поведінку, яка дозволяє однозначно задати послідовність виконання операцій відповідно до CEI:

- Перевірка (Checks) передбачає, що на початку ми виконуємо перевірку умов того, чи може користувач поставити токени на стейкінг: перевіряється, що бажана сума (amount) більша або дорівнює мінімальній сумі стейкінгу (minStakeAmount), користувач має достатній баланс токенів на гаманці. Перевірка представлена як дія CheckConditions в алгебраїчних специфікаціях (представлена далі у тексті).
- Ефект (Effects) грунтується на тому, що на цьому етапі відбувається оновлення внутрішнього стану контракту та користувача: збільшуємо значення кількості токенів на стейкінгу для даного користувача, оновлюємо значення загальної кількості токенів на стейкінгу в даному смарт-контракті. Ця фаза важлива для запобігання вразливостям, оскільки всі внутрішні зміни стану відбуваються до будь-яких зовнішніх взаємодій. Представлена як дія UpdataState в алгебраїчних специфікаціях (представлена далі у тексті).
- Взаємодії (Interactions) полягає у реалізації процесу передачі активів, коли токени переказуються з гаманця користувача на рахунок смарт-контракту. Розміщення цього кроку в кінці гарантує, що будь-які зовнішні виклики (як-от transferFrom у Solidity) виконуються лише після повного та безпечного оновлення внутрішнього стану контракту. Представлена дією Transfer.

Кожна підповедінка відповідає набору дій, які можуть виконуватися агентами. Реалізація поведінки Stake:

```
user_s:User
contract_r:Contract


 InitState = Stake_Beh(user_s, contract_r, 100),

 Stake_Beh(user,contract,amount)=(
  (stakingCheckConditions(user, contract, amount);(
        stakingUpdataState(user,amount);
        stakingTransfer(contract,amount)
   ) + !stakingCheckConditions(user, contract, amount)
  ))
....
```

де

`InitState` – це початковий стан для поведінки `Stake_Beh`.

Дія stakingCheckConditions в алгебраїчній формі представлена наступним чином:

```
stakingCheckConditions(user, contract, amount)=(
 (amount >= contract.minStakeAmount && user.balance >= amount) ->
 <'Staking is allowed'>
 (1)
),
```

Якщо кон’юнкція передумови та стану середовища виконується, тоді ми переходимо до наступної дії `UpdataState`, а стан системи не змінюється оскільки дія має (1) в постумові.

Дія stakingUpdataState в алгебраїчній формі:

```
stakingUpdataState(user,contract, amount)=(
 (1)->
 <'Update Sender Balance'>
 (user.stakedToken = user.stakedToken + amount;
  contract.totalStaked = contract.totalStaked + amount;
  user.lastRewardTime = contract.currentTime)
),
```

де (1) в передумові означає, що дія виконується будь-яких сумісних станів середовища.

Дія stakingTransfer в алгебраїчній формі:

```
stakingTransfer(user, contract, amount)=(
 (1)->
 <'Update Receiver Balance/Transfer'>
 (user.balance = user.balance - amount;
       contract.totalStaked = contract.totalStaked + amount)

)
```

Однією з ключових складностей використання формальних моделей для генерації смарт-контрактів є те, що великі мовні моделі не завжди здатні правильно зіставити формальні описи з відповідними шаблонами реалізації мовою Solidity. Оскільки LLM навчаються на відкритих даних, вони можуть не мати доступ до великої кількості прикладів формальних моделей, створених з

використанням інсерційного моделювання, що ускладнює для моделей розуміння їхньої структури та семантики. Відсутність достатньої кількості прикладів формальних моделей у навчальних даних обмежує здатність LLM адаптувати ці описи до реальних шаблонів програмування. Це, в свою чергу, знижує точність і надійність згенерованого коду.

Для того, щоб використовувати формальну модель смарт-контракту для генерації коду, ми надали обраним LLM приклади алгебраїчних специфікацій та відповідний код Solidity. Це дозволяє моделям краще зрозуміти, як формальні описи (наприклад, передумови, постумови чи дії) переводяться в конкретні конструкції мови Solidity.

Такий підхід називається промптингом з невеликою кількістю прикладів – процес надання моделі штучного інтелекту кількох зразків у запиті, для того щоб модель краще розуміла задачу та генерувала відповідні результати ("Few-Shot Prompting", 2025). Промптинг з невеликою кількістю прикладів – це частина навчання в контексті – здатності моделі адаптувати свою поведінку на основі прикладів, наданих у вхідному запиті (Cleary, 2025). Тобто, дослідник формує запит до великої мовної моделі, що містить кілька демонстраційних прикладів «вхід → очікуваний вихід», після чого додає новий запит без відповіді. Ключова ідея полягає в тому, що модель розпізнає закономірності між прикладами та узагальнює їх на новий випадок. Цей метод є корисним у сценаріях, коли немає великої кількості навчальних даних як в нашому випадку. У контексті генерації смарт-контрактів кожний демонстраційний блок складається з алгебраїчної специфікації контракту та відповідного фрагмента коду на Solidity.

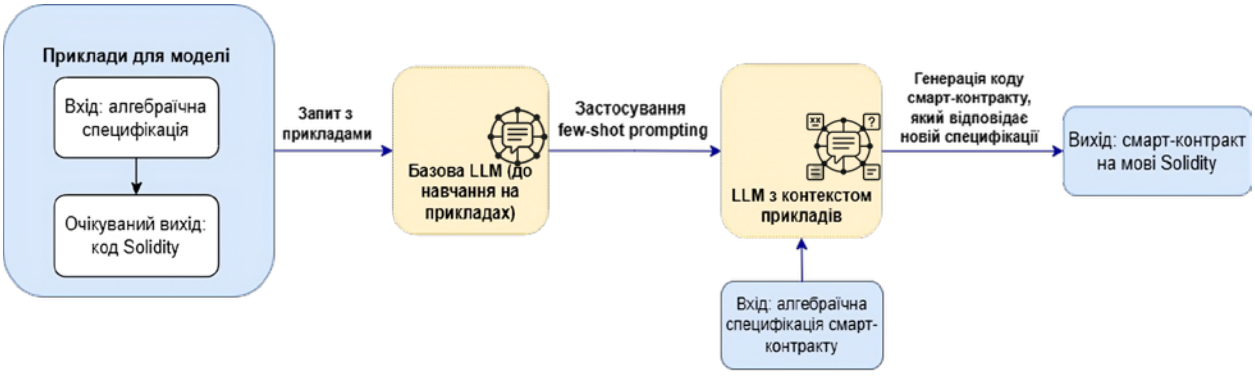

Рис. 2. Промптинг з невеликою кількістю прикладів для генерації смарт-контрактів

Ми використовували приклади, які демонструють патерни, логічно подібні до тих, що представлені в нашій моделі: стейкінг, розрахунок винагород, часові обчислення, але з іншими параметрами чи логікою. Варто зазначити, що надмірна кількість прикладів може перевантажити контекстне вікно LLM (Bergmann, 2024) при використанні запиту із кількома прикладами, що знижує продуктивність генерації. Оскільки в сценаріях стейкінгу ми використовуємо стандартні патерни (наприклад, Перевірки – Ефекти – Взаємодії, події, перевірки), тому 5–8 прикладів достатньо, щоб LLM засвоїла ці структури без перевантаження контексту.

Спочатку нами було надано приклади, які містили просту відповідність між діями з формальної моделі (наприклад, перевірка умов або переведення коштів) та відповідним фрагментом коду Solidity. Це дозволяє моделі правильно транслювати передумови та постумови у програмні конструкції.

Також у прикладах ми додавали пояснення, щоб LLM точно зіставляла конструкції формальної моделі з Solidity. Наприклад, в дії CheckConditions(user, contract, amount) user – це ініціатор дії, тобто msg.sender, contract – сам смарт-контракт, а contract.minContribution – це змінна стану цього контракту.

Наприклад:

Дія в алгебраїчній формі	Відповідний код Solidity
CheckConditions(user, contract, amount) = ((amount >= contract.minContribution && user.balance >= amount) -> <'Action is allowed'> (1))	require(amount >= minContribution, "Below minimum contribution"); require(token.balanceOf(msg.sender) >= amount, "Insufficient token balance");

UpdateState(user,contract, amount) =
(
(1) ->
<'Update Sender Balance'>
(user.stakedToken = user.stakedToken + amount;
contract.totalStaked = contract.totalStaked + amount;
)
)
	User storage user = users[msg.sender];
	user.stakedToken += amount;
	totalStaked += amount;

Transfer(user, contract, amount)=((1)-> <'Transfer tokens'> (user.balance = user.balance - amount; contract.balance = contract.balance + amount))	token.transferFrom(msg.sender, address(this), amount);

Оскільки в моделі патерн Перевірки – Ефекти – Взаємодії (CEI), який використовується для функцій stake та withdraw, у нас представляється трьома окремими діями, нам також необхідно надати LLM приклад, який демонструє об'єднання цих трьох дій в одну функцію в коді Solidity. Для цього ми наводимо приклад поведінки та відповідної функції у коді Solidity:

Поведінка для патерну CEI	Відповідна функція в коді Solidity

CEI_Beh(user,contract,amount)=(
(CheckConditions(user,contract, amount);(
UpdateState(user, contract, amount);
Transfer(user,contract,amount)) +
!CheckConditions(user,contract, amount)
))
	function CEI(uint256 amount) external nonReentrant {
	//CheckConditions
	require(amount >= minContribution, "Below minimum contribution");
	require(token.balanceOf(msg.sender) >= amount, "InsufficientTokenBalance");
	
	//UpdateState
	User storage user = users[msg.sender];
	user.stakedToken += amount;
	totalStaked += amount;
	
	//Transfer
	token.transferFrom(msg.sender, address(this), amount);
	emit Transferred (msg.sender, amount);
	}

Ми також надали приклад повного смарт-контракту для простої моделі, щоб продемонструвати, як у коді визначаються: структури, співставлення та інші конструкції мови Solidity відповідно до повної формальної моделі смарт-контракту.

Після контекстного налаштування запиту методом промптингу з невеликою кількістю прикладів, ми виконували генерацію коду смарт-контракту з використанням обраних великих мовних моделей, використовуючи алгебраїчні специфікації як основу для формулювання запиту.

Таблиця 1

Кількісна оцінка результатів генерації смарт-контрактів

Модель/ інструмент	Тип запиту	Кількість успішно скомпільованих смарт-контрактів	Кількість смарт-контрактів, що повністю відповідали вимогам	Середня кількість вразливостей
ChatGPT-4	Природномовний запит	5/5	3/5	2.2
	Формалізований запит	5/5	5/5	0.8
Claude 3.7 Sonnet	Природномовний запит	3/5	2/5	3.0
	Формалізований запит	4/5	4/5	1.0
DeepSeek-V3	Природномовний запит	4/5	3/5	2.4
	Формалізований запит	5/5	4/5	0.7

Оцінка згенерованих смарт-контрактів:

- Синтаксична коректність: згенеровані контракти містили незначні неточності, такі як пропущені чи зайві дужки у виразах, посилання на функції чи змінні, яких не існує у цій версії коду. Також виникали помилки через використання невірних типів даних (наприклад, ‘int’ замість ‘uint256’), неправильне визначення структур. Але надання додаткових простих прикладів дозволило покращити генерацію та усунути подібні синтаксичні помилки. Загалом синтаксична коректність залежить від якості навчання LLM, точності запиту та актуальності використаних бібліотек.
- Логічна відповідність вимогам: згенеровані контракти правильно реалізують логіку, відповідно до вимог (правильний розрахунок винагород, розблокування токенів для користувачів, перевірки умов).
- Відсутність вразливостей: при перевірці згенерованих смарт-контрактів за допомогою інструменту Slither критичні вразливості не було виявлено. Проте деякі контракти містили недоліки, такі як неефективність (надмірне споживання газу) або наявність мертвого коду, які можуть виникати через особливості оптимізаційних рішень LLM або специфіку перетворення формальної моделі в код.

Таблиця 2

GitHub Copilot	Природномовний запит	2/5	2/5	3.5
	Формалізований запит	4/5	4/5	1.3

Як видно з таблиці, незалежно від конкретної LLM, використання алгебраїчних специфікацій стабільно покращує результати: зростає частка відповідності вимогам, а середня кількість виявлених вразливостей у коді помітно зменшується. Наведена вибірка дає змогу виявити типові тенденції та здійснити порівняння підходів, проте для узагальнення висновків на широкий клас смарт-контрактів у подальших дослідженнях доцільно збільшити кількість прикладів щонайменше до кількох десятків на кожну конфігурацію.

Результати експерименту засвідчили, що контракти, згенеровані на основі формальної моделі, продемонстрували вищий рівень відповідності початковим вимогам. Формальний опис забезпечив вищу повноту реалізації запланованих функцій, тоді як текстовий опис вимог призвів до відсутності деяких елементів, таких як, наприклад, механізм управління пулом винагород, або містив неточності в реалізації розрахунку винагород. У контексті безпеки контракти, створені на основі формальної специфікації, показали кращі результати, оскільки вони містили більше перевірок стану. Однак, варто зазначити, що великі мовні моделі не завжди реалізують вимоги безпеки (наприклад, використання бібліотеки ReentrancyGuard) та оптимізації для мінімізації споживання газу, якщо це явно не прописано у запиті.

Таблиця 3

Порівняння результатів генерації смарт-контрактів за допомогою ШІ

Параметр	Текстовий запит	Запит з алгебраїчними специфікаціями
Синтаксична коректність	Згенеровані контракти містили незначні неточності.	Згенеровані контракти містили незначні неточності.
Відповідність специфікаціям	Реалізує основні функції, але може допускати помилки в реалізації складної логіки через неправильну інтерпретацію.	Формальні моделі забезпечують вищу точність і повноту, але вимагають правильної інтерпретації з боку LLM, що потребує спеціалізованого контекстного налаштування за допомогою промптингу з невеликою кількістю прикладів.
Безпека	Середній рівень безпеки, що залежить від якості реалізації LLM; потрібні додаткові перевірки. Без додаткових зазначень у запиті, може не дотримуватися строгих безпекових шаблонів, таких як контроль над переповненням або повторним викликом (reentrancy). Згенеровані контракти можуть бути схильні до вразливостей, як-от неправильний порядок операцій при розрахунку винагород ("ділення перед множенням"), що призводить до втрати точності, або вразливості до повторного входу через порушення шаблону "Checks-Effects-Interactions".	Середній рівень безпеки, потребує перевірки отриманого коду. Безпека залежить від інтерпретації LLM. Формальні моделі не гарантують абсолютної безпеки без верифікації. Проте використання алгебраїчних специфікацій дозволяє зменшити ймовірність деяких типових вразливостей, таких як: – ділення перед множенням (оскільки порядок обчислень чітко визначається у формулі); – повторний виклик (через явне формальне задання змін стану до зовнішніх викликів). – неправильну обробку умов тощо.
Узагальнення	Підходить для швидких прототипів, але менш надійний. Потребує додаткової верифікації.	Кращий для критичних систем або систем зі складною логікою, але потребує також потребує додаткової перевірки безпеки.

Обмеження використаних моделей у контексті генерації смарт-контрактів:

- Через механізм автодоповнення великі мовні моделі у деяких випадках схильні самостійно доповнювати невизначені або неповні частини логіки, що може призвести до небажаної поведінки у смарт-контракті.
- LLM забезпечує генерацію коду на основі логіки та патернів контрактів з відкритих репозиторіїв. Це означає, що якість коду залежить від якості навчальних даних. Також згенерований код може базуватися на застарілих версіях Solidity або бібліотек, якщо модель не оновлена.
- Моделі не завжди виявляють або запобігають типовим вразливостям (повторний виклик, переповнення цілих чисел, доступ без авторизації тощо).

Висновки

У цій статті представлено результаті порівняння двох підходів до генерації коду смарт-контрактів: на основі текстового опису та на основі формальної моделі. Було описано приклад використання промптингу з невеликою кількістю прикладів у контексті генерації смарт-контрактів. У ході роботи було проаналізовано смарт-контракти, згенеровані великими мовними моделями (ChatGPT-4, Claude 3.7 Sonnet, DeepSeek-V3) та інструментом на основі штучного інтелекту GitHub Copilot, з оцінкою їхньої синтаксичної коректності, відповідності початковим вимогам та безпеки.

Результати аналізу підтверджують, що формальні специфікації сприяють більш точній реалізації вимог завдяки чіткому визначенню стану, дій і поведінки. Отже, гіпотеза частково підтверджується: формальні моделі забезпечують вищу точність, але не гарантують вищої безпеки через можливі помилки інтерпретації LLM або генерації неефективного коду (надмірне споживання газу, мертвий код). Для досягнення високої безпеки потрібні додаткові механізми верифікації.

Текстові запити простіші для швидкої генерації, але менш надійні та більш схильні до генерації прихованих вразливостей через свою двозначність, тоді як формальні моделі потребують додаткових зусиль для усунення помилок інтерпретації LLM. Важливим аспектом є також зменшення семантичного розриву між вимогами та реалізацією при використанні формальних моделей. Чітко визначені передумови та постумови для кожної операції усувають можливість неправильного розуміння логіки мовними моделями, які досить розповсюджені при роботі з текстовими описами.

Також нами було проаналізовано обмеження використання LLM, які впливають на якість згенерованого коду смарт-контрактів та коректність реалізації логіки.

Результати цього дослідження обмежені вибором конкретних великих мовних моделей та орієнтовані на генерацію смарт-контрактів мовою Solidity для платформи Ethereum. Саме тому, отримані висновки можуть не бути повністю узагальненими для інших мов програмування смарт-контрактів чи для інших архітектур LLM.

Важливо також зазначити, що навіть при використанні формальних моделей процес генерації смарт-контрактів повинен включати етап ретельного тестування та аудиту, щоб забезпечити їхню надійність у реальних умовах.

СПИСОК ЛІТЕРАТУРИ

Bergmann, D. (2024). Context window. IBM.com. https://www.ibm.com/think/topics/context-window

Chatterjee, S., Ramamurthy, B. (2025). Efficacy of Various Large Language Models in Generating Smart Contracts. In: Arai, K. (eds) Advances in Information and Communication. FICC 2025. Lecture Notes in Networks and Systems, vol 1284. Springer, Cham. https://doi.org/10.1007/978-3-031-85363-0_31

Cleary, D. (2025). In Context Learning Guide. PromptHub. https://www.prompthub.us/blog/in-context-learning-guide

Crytic. Slither static analysis framework. GitHub. https://github.com/crytic/slither

Fadi, B., Napoli, E. A., Gatteschi, V., & Schifanella, C. (2024). Automatic Smart Contract Generation Through LLMs: When The Stochastic Parrot Fails. In Proceedings of DLT 2024. CEUR.

Fravoll. Checks, Effects, Interactions pattern in Solidity. (2025) https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html

Kang, I., Woensel, W. V., & Seneviratne, O. (2024). Using Large Language Models for Generating Smart Contracts for Health Insurance from Textual Policies. In AI for Health Equity and Fairness (pp. 129–146). Springer. https://doi.org/10.1007/978-3-031-63592-2_11

Letichevsky, A., Kapitonova, J., Letichevsky Jr, A., Volkov, V., Baranov, S., & Weigert, T. (2005). Basic protocols, message sequence charts, and the verification of requirements specifications. Computer Networks, 49(5), 661–675. https://doi.org/10.1016/j.comnet.2005.05.005

Letichevsky, A., Letychevskyi, O., & Peschanenko, V. (2016). Insertion modeling and its applications. Computer Science Journal of Moldova, 72(3), 357–370.

Modi, M. (2024). Transforming software development through generative AI: A systematic analysis of automated development practices. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 10(6), 536–547. https://doi.org/10.32628/cseit24106197

Peschanenko, V., Poltorackiy, M., & Konnova, O. (2025). Verification of Smart Contract Code Generated by Applying Artificial Intelligence. In: Ermolayev, V., et al. Information and Communication Technologies in Education, Research, and Industrial Applications. ICTERI 2024. Communications in Computer and Information Science, vol 2359. Springer. https://doi.org/10.1007/978-3-031-81372-6_1

Prompt Engineering Guide. Few-shot prompting. https://www.promptingguide.ai/techniques/fewshot

Wong, M. F., Guo, S., Hang, C. N., Ho, S., & Tan, C. W. (2023). Natural language generation and understanding of big code for AI-assisted programming: A review. Entropy, 25.

COMPARATIVE ANALYSIS OF SOLIDITY SMART CONTRACT GENERATION USING LARGE LANGUAGE MODELS BASED ON FORMAL ALGEBRAIC SPECIFICATIONS

Volodymyr Peschanenko[1], Olha Konnova[2]

[1, 2] Kherson State University,
Department of Computer Science and Software Engineering, Kherson, Ukraine
[1] E-mail: vladim@ksu.ks.ua, ORCID: 0000-0003-1013-9877
[2] E-mail: okonnova@ksu.ks.ua, ORCID: 0000-0001-5590-9527

This article presents a comparative analysis of automatic Solidity smart contract generation using large language models (LLMs) based on two approaches: natural language textual descriptions and formal algebraic specifications. The study analyzes smart contracts generated by LLMs (ChatGPT-4, Claude 3.7 Sonnet, DeepSeek-V3) as well as by the AI-based tool GitHub Copilot, evaluating their syntactic correctness, compliance with initial requirements, and security. The security of the smart contracts was assessed through static analysis using the Slither tool to identify potential vulnerabilities, including reentrancy attacks and arithmetic overflows.

To create the formal specification, we applied the Insertion Modeling System (IMS), which enables formal descriptions of smart contract behavior using algebraic specifications. To improve the interpretation of these specifications by large language models, we proposed a few-shot prompting approach, which involves providing LLM with demonstration examples of algebraic specifications and the corresponding Solidity code to improve the interpretation of formal models. The proposed approach allowed us to increase the accuracy of translating formal specifications into code, reduce the number of syntax errors, and ensure better compliance with the original model.

The study found that natural language prompts are easier to use for quick code generation but are less reliable, as their inherent ambiguity tends to produce hidden vulnerabilities. In contrast, formal models require additional effort to guide the LLM and eliminate interpretation errors, but they allow for a much more accurate representation of contract logic, resulting in higher-quality generation of contracts with complex behavior.

Keywords - blockchain, smart contracts, artificial intelligence, LLM, algebraic specifications, insertion modeling, smart contract generation, Solidity