УДК 004.414.23
DOI https://doi.org/10.32782/2663-5941/2025.4.2/28

Нікітченко М.І.
Національний університет «Одеська політехніка»

АРХІТЕКТУРА ТА РЕАЛІЗАЦІЯ ПЛАГІНА ІНКРЕМЕНТАЛЬНОЇ ВАЛІДАЦІЇ UML-МЕТАМОДЕЛЕЙ З ДВОМА ПОДАННЯМИ

У статті розглянуто проблему забезпечення узгодженості UML-моделей у процесі розробки складних програмних систем, актуальність якої зумовлена високою вартістю виявлення архітектурних дефектів на пізніх етапах життєвого циклу. Сучасні CASE-інструменти моделювання пропонують або гнучкість формату JSON для статичних структур з низьким рівнем формалізму, або строгість формату XMI, що важко інтегрується у повсякденну розробку через складність обробки. Водночас жоден із цих підходів не забезпечує повного поєднання переваг гнучкості й формалізму, що створює потребу в розробці гібридного рішення.

Метою роботи є представлення архітектури та практичної реалізації плагіна, який інтегрує два формати представлення UML-моделей – легковагий JSON для опису статичної структури та формально строгий XMI для опису динаміки системи. Особливістю плагіна є застосування спеціалізованого UML-профілю для фізичної реалізації відношення відповідності між цими поданнями, що дозволяє здійснювати швидку інкрементальну валідацію через граф залежностей. Методологія включає поєднання синтаксичної перевірки JSON Schema, семантичної інкрементальної валідації та глибокої формальної верифікації транзитивних властивостей за допомогою SAT-аналізу.

Результати експериментальної оцінки показали, що запропонований плагін значно ефективніший за існуючі аналоги, оскільки інкрементальна валідація зменшує час перевірки порівняно з повною перевіркою моделей. Таким чином, забезпечується можливість ефективної інтеграції у конвеєри безперервної інтеграції та доставки,, що робить інструмент придатним для використання у режимі реального часу. Висновки дослідження підкреслюють перспективність гібридного підходу до зберігання UML-моделей та визначають напрями для подальшого розвитку, включаючи підтримку інших мов програмування та розширення функціональності плагіна.

Ключові слова: *UML, узгодженість моделей, інкрементальна валідація, JSON Schema, XMI, граф залежностей, SAT-аналіз.*

Постановка проблеми. Сучасна програмна інженерія стикається з постійним зростанням складності систем, що вимагає ефективних підходів до моделювання їхньої архітектури. В контексті Model-Driven Engineering розроблено багато методик для забезпечення консистентності артефактів [1], але на практиці часто виникає поступове розходження між проектною моделлю і документацією, та її реальною імплементацією у вихідному коді. В результаті, цей розрив породжує клас прихованих архітектурних дефектів – семантичних помилок, які не виявляються стандартними компіляторами чи валідаторами. Прикладами можуть бути виклик методу, що був перейменований у коді, але зберіг стару назву в діаграмі послідовності, або використання застарілого типу даних у поведінковому сценарії, або ж навіть суперечності між діаграмами [2]. Виявлення таких дефектів на пізніх етапах життєвого циклу є на порядки дорожчим, що робить проблему забезпечення консистентності ключовим фактором економічної ефективності розробки.

В основі цієї проблеми лежить фундаментальний вибір формату для зберігання UML-моделей. Жоден з існуючих стандартів не надає універсального рішення, змушуючи команди робити компроміс між формальною строгістю та гнучкістю розробки.

XMI (XML Metadata Interchange) є офіційним стандартом OMG, що гарантує максимальну повноту та формальну точність опису, що особливо критично для складних поведінкових діаграм (станів, послідовностей, діяльності), а також для забезпечення сумісності з існуючими CASE-інструментами та можливість застосування формальних обмежень мовою OCL. Однак його надлишковий синтаксис робить XMI-файли громіздкими, складними для ручного аналізу та

обробки в системах контролю версій, таких як Git, де навіть незначні структурні зміни призводять до значних конфліктів злиття.

JSON (JavaScript Object Notation), навпаки, є легковагим, текстовим і зручним для розробників форматом, що ідеально підходить для опису статичних структур: класів, атрибутів, пакетів. Його простота забезпечує швидку обробку разом з меншими затратами пам'яті та легку інтеграцію з інструментами розробки. Проте JSON не має вбудованих механізмів для представлення складної поведінкової семантики, крос-посилань між елементами та формальних обмежень, що є сильною стороною XMI. Саме тому спроба описати динаміку системи в чистому JSON призводить або до втрати семантики, або до створення переобтяжених документів.

Виникає глибокий розрив між теоретичними моделями узгодженості, що вимагають формалізму, та реальними інструментами, які використовуються в щоденній практиці розробки. Існуючі CASE-засоби змушують команди робити компромісний вибір між формалізмом та гнучкістю, не пропонуючи нативного рішення для гібридного зберігання моделей, яке б поєднувало найкращі риси обох світів.

Ключова проблема полягає в тому, що виявлення цих семантичних розривів на пізніх етапах життєвого циклу (під час інтеграційного тестування або, що найгірше, вже в експлуатації) є на порядки дорожчим, ніж на етапі проектування. Тому проблема забезпечення консистентності є не лише технічним питанням якості, а й ключовим фактором економічної ефективності та керованості процесу розробки.

Аналіз останніх досліджень і публікацій. Для обґрунтування актуальності та новизни запропонованого рішення було проведено детальний аналіз сучасних CASE-засобів та форматів зберігання UML-моделей [3, 4]. Він показав, що ринок інструментів моделювання фактично розділився на два полюси: зручні для розробника, але формально слабкі інструменти, що використовують JSON, та формально строгі, але громіздкі й негнучкі інструменти, що базуються на XMI. Тим не менш, жоден із них не пропонує комплексного рішення для одночасної підтримки гнучкості та семантичної повноти.

Ця робота є прямим практичним втіленням та емпіричною перевіркою теоретичних засад, викладених у серії попередніх статей. Вона завершує цикл досліджень, переводячи абстрактні концепції в площину конкретного програмного інструменту. Перша робота формалізувала метамодель з двома поданнями, а також було введено систему формальних інваріантів для перевірки узгодженості моделі [5]. У другій роботі було деталізовано практичну реалізацію структурного подання через JSON Schema [6]. Було визначено набір інваріантів для валідації статичної структури, частина з яких реалізується безпосередньо засобами схеми (наприклад, перевірка типів, обов'язкових полів), а частина (наприклад, ациклічність) потребує зовнішніх валідаторів. У третій роботі було розроблено спеціалізований UML-профіль для поведінкового подання, який дозволяє фізично реалізувати відношення зв'язку через теги в XMI [7]. Також було представлено теоретичний алгоритм інкрементальної валідації, адже інкрементальне поширення змін є ефективнішим за повну повторну трансформацію моделей, особливо для великомасштабних систем [8].

Аналіз існуючих рішень та теоретичних робіт дозволяє виділити ключові проблеми, які вони не вирішують комплексно.

– **Відсутність нативної мультиформатної архітектури:** Жоден з інструментів не побудований на ідеї одночасного використання JSON для структури та XMI для поведінки як основи для роботи з єдиною, цілісною моделлю. Інструменти або обирають один формат, або пропонують ручну конвертацію.

– **Неефективність валідації:** Перевірка узгодженості, якщо вона взагалі є, виконується для всієї моделі. Ідея ефективної інкрементальної валідації на основі графа залежностей, що охоплює лише змінені частини та їхній вплив, не реалізована в аналогах, що робить їх непрактичними для великих проектів у режимі реального часу.

– **Слабкий зв'язок «модель-код»:** Існуючі інструменти є переважно засобами візуального моделювання, відірваними від вихідного коду. Вони не забезпечують безперервну, фонову валідацію моделі, що автоматично генерується з коду, як це пропонується в плагіну.

– **Складність інтеграції в автоматизовані процеси:** Відсутність простого та потужного Command Line Interface (CLI) ускладнює вбудовування перевірки консистентності в автоматизовані конвеєри збірки та тестування.

В результаті, ринок інструментів моделювання фактично розділився на два незадовільні полюси: зручні для розробника, але формально слабкі інструменти, та формально строгі, але незручні для розробника інструменти. Це створює значну нішу для інструменту, який би синтезував сильні

сторони обох підходів. Розроблений плагін є саме таким архітектурним рішенням, що поєднує зручність JSON з формалізмом XMI через потужний фокус на узгодженості.

Постановка завдання. Метою статті є представлення архітектури, практичної реалізації та експериментальної оцінки розроблюваного плагіна, що є програмним втіленням розробленої раніше теоретичної моделі та призначений для розв'язання вищезазначених проблем розриву між гнучкістю розробки та формальною строгістю моделювання.

Для досягнення цієї мети поставлено наступні завдання:

– описати високорівневу архітектуру плагіна, яка реалізує теоретичну метамодель через модульну структуру та продемонструвати потік даних від вихідного коду до артефактів моделі;

– продемонструвати ключові механізми реалізації, в тому числі автоматичну побудову внутрішньої моделі з вихідного коду Python, фізичну реалізацію відношення відповідності μ через спеціалізований UML-профіль та теги JsonRef у XMI, що забезпечує трасування між поданнями, а також практичну роботу інкрементальної валідації на основі графа залежностей G_{MB}, що розрізняє структурні, семантичні та трасувальні зв'язки;

– навести інструкцію з використання плагіна та продемонструвати його роботу на контрольному прикладі;

– провести експериментальну оцінку продуктивності, вимірявши час повної та інкрементальної валідації, а також використання пам'яті, і порівняти отримані результати з аналогами.

Виклад основного матеріалу. В основі плагіна лежить формальна UML-метамодель $M = (M_S, M_B, \mu)$, детально описана в [5]. Ця модель свідомо розділяє єдину концептуальну модель на два подання, щоб поєднати переваги різних форматів і вирішити фундаментальний конфлікт між гнучкістю та строгістю.

– Структурне подання (M_S): Це опис статичної архітектури системи (класів, атрибутів, операцій, пакетів тощо). Воно зберігається у форматі JSON.

– Поведінкове подання (M_B): Це опис динаміки системи (діаграм станів, послідовностей, діяльності). Воно зберігається у стандартному для UML форматі XMI.

– Відношення відповідності (μ): Це ключовий механізм, що слугує зв'язком між двома поданнями. Воно формально пов'язує елементи поведінки з їхніми структурними аналогами.

Така архітектура дозволяє виконувати ефективну інкрементальну валідацію. Після зміни у вихідному коді, яка впливає на M_S, система може швидко визначити через μ та граф залежностей G_{MB} вузьке коло елементів в M_B, які потребують повторної перевірки, замість того, щоб аналізувати всю модель цілком.

Сам плагін побудований на основі модульної архітектури, розробленої для чіткого розподілу відповідальності та для забезпечення високої розширюваності. Центральним елементом архітектури є внутрішня модель: набір Python-об'єктів (dataclasses), що представляють елементи UML в пам'яті. Ця модель слугує проміжним шаром, який повністю ізолює компоненти один від одного.

Нижче наведено модулі системи, згідно зі структурою проекту.

Модуль *parsers* відповідає за аналіз вихідного коду.

Модуль *core* містить ядро системи. В його складі можна виділити внутрішні моделі даних, реалізацію графа залежностей та логіку персистентності.

Модуль *generators* відповідає за генерацію моделей, як для структурної частини, так і для поведінкової.

Модуль *validators* містить такі компоненти як валідаційний движок та зв'язок із SAT-вирішувачем.

Модуль *integration* забезпечує зовнішню інтеграцію за рахунок CLI та IDE API.

Потік даних у системі організовано наступним чином. Спершу плагін отримує вихідний код на мові Python і для кожного файлу будує абстрактне синтаксичне дерево (Abstract syntax tree, AST) за допомогою вбудованого модуля ast.

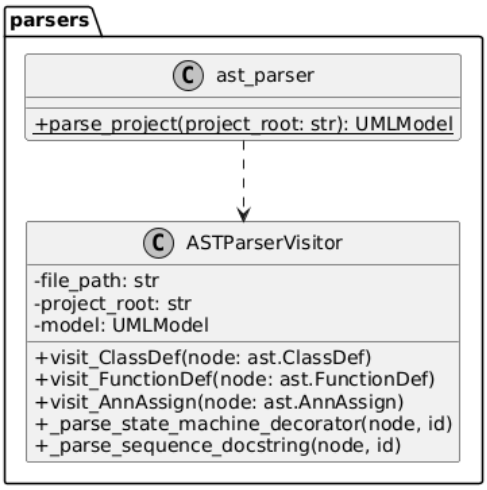

Рис. 1. Модуль parsers

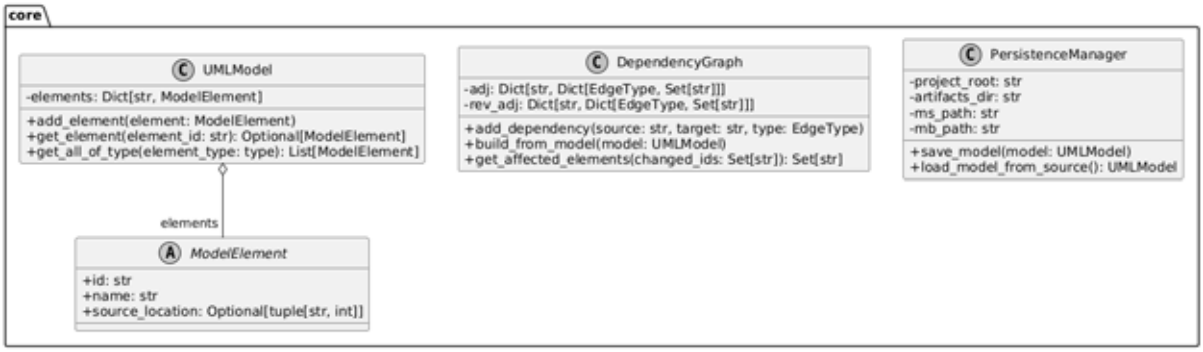

Рис. 2. Модуль core

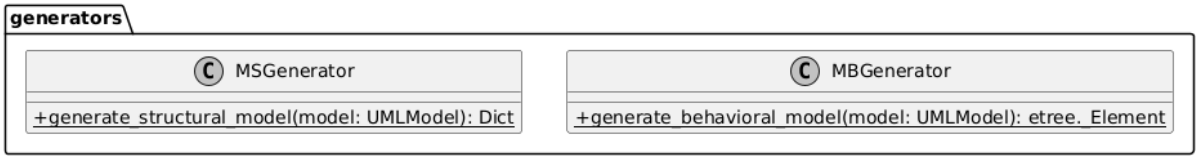

Рис. 3. Модуль generators

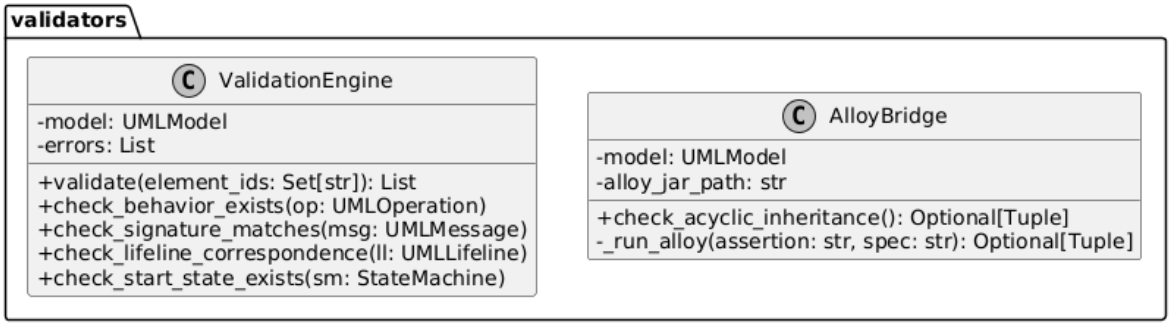

Рис. 4. Модуль validators

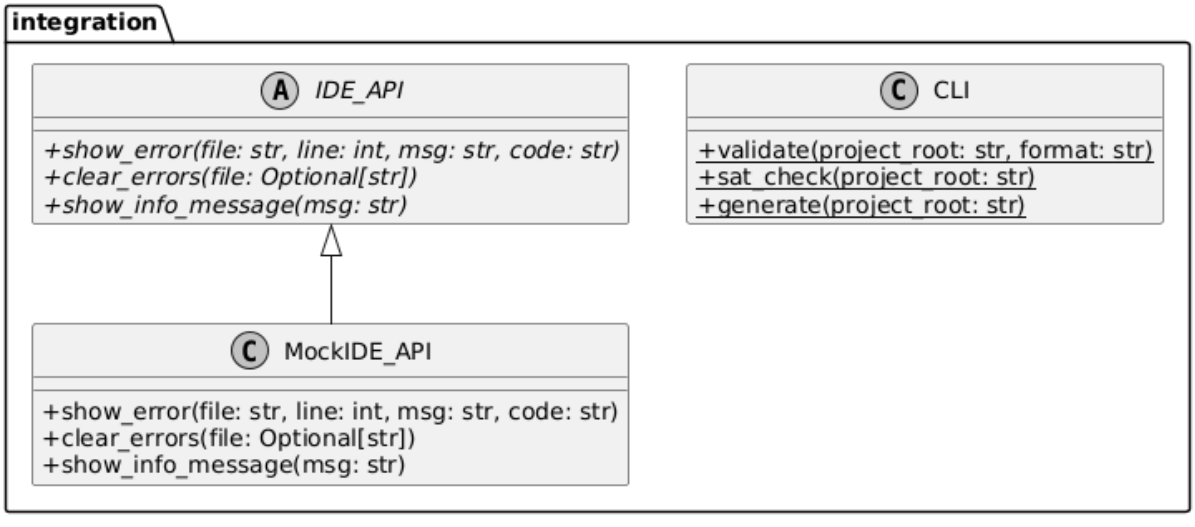

Рис. 5. Модуль integration

Потім спеціалізований клас ASTParserVisitor рекурсивно обходить AST. При відвідуванні кожного вузла (наприклад, ast.ClassDef для класів, ast.FunctionDef для методів) він витягує необхідну інформацію та створює екземпляри внутрішніх моделей даних – об'єктів з core/models.py. Одночасно з наповненням внутрішньої моделі будується граф залежностей G_{MB}, який фіксує всі структурні, семантичні та трасувальні зв'язки між елементами моделі. Генератори (ms_generator, mb_generator) обходять наповнену внутрішню модель і серіалізують її у фінальні файли: model.struct.json та model.behav.xmi, які зберігаються в директорії.uml-consistency/.

Внутрішня модель, реалізована на основі dataclasses, є архітектурним стрижнем плагіна. Вона діє як шар абстракції, що повністю роз'єднує логіку аналізу вихідного коду від логіки генерації цільових форматів. Наприклад, ASTParserVisitor знає лише, як перетворити вузли AST у ці dataclasses, і нічого не знає про JSON чи XMI. Аналогічно, генератори працюють виключно з цими об'єктами, не маючи уявлення про Python чи AST. Таке роз'єднання забезпечує високу розширюваність: для підтримки нової мови програмування (наприклад, Java) достатньо буде реалізувати новий модуль-парсер, який би наповнював ту саму внутрішню модель, не змінюючи решту системи.

В якості основної мови розробки було обрано Python як через його динамічну природу, потужні засоби інтроспекції та, що найголовніше, наявність вбудованого модуля ast, який є ідеальною основою для статичного аналізу коду. Для генерації XMI використовується галузевий стандарт – бібліотека lxml, а для валідації JSON – jsonschema.

Робота плагіна починається з глибокого аналізу коду. ASTParserVisitor реалізує патерн Visitor для обходу дерева AST. Наприклад, при відвідуванні вузла ast.ClassDef з декоратором @state_machine(initial_state=States.IDLE), парсер не лише створює об'єкт UMLClass для самого класу, але й пов'язаний з ним об'єкт StateMachine, встановлюючи посилання json_id на ідентифікатор класу. Аналогічно, методи класу з декоратором @transition(source=…, target=…) трансформуються в об'єкти UMLTransition, а структуровані docstring-коментарі з тегом @sequence парсяться для створення діаграм послідовностей.

Також було виконано реалізацію відношення μ, яка є ключовою інновацією, що забезпечує зв'язність моделі.

Нижче наведено приклад згенерованого XMI-фрагменту, де лінія життя UMLLifeline посилається на клас Device зі структурного подання:

```
<packagedElement xmi:type=»uml:Lifeline»
xmi:id=»seq1.lifeline.device» name=»self»>
<xmi:Extension extender=»UMLProfile»>
<JsonRef jsonId=»my_project.Device»/>
</xmi:Extension>
</packagedElement>
```

Тег JsonRef, що є частиною розробленого UML-профілю, є фізичним втіленням абстрактного відношення μ, що дозволяє валідатору під час аналізу XMI-файлу швидко (з часовою складністю O(1) за допомогою хеш-таблиці) отримати доступ до повного опису класу Device з JSON-файлу.

Водночас ефективність інкрементальної валідації залежить від точності графа залежностей. На відміну від простого графа, реалізація в dependency_graph.py та теоретична модель розрізняють три типи ребер, що дозволяє виконувати більш точний аналіз впливу змін:

– структурні: Відображають відношення володіння (наприклад, від UMLClass до UMLOperation);

– семантичні: Фіксують логічні зв'язки (наприклад, від UMLMessage до UMLOperation, яку воно викликає);

– трасувальні: Реалізують відношення μ між поданнями (наприклад, від UMLLifeline (M_B) до UMLClass (M_S)).

В результаті, подібна класифікація дозволяє валідатору виконувати цільові перевірки. Наприклад, зміна сигнатури методу в коді поширюється через семантичне ребро до відповідного UMLMessage і запускає перевірку інваріанта SignatureMatches. Тому така точність запобігає виконанню зайвих перевірок і є ключем до високої продуктивності.

Для забезпечення балансу між швидкістю відгуку та глибиною аналізу, плагін реалізує трирівневу стратегію валідації, де наступний рівень є більш обчислювально складним, але й більш потужним.

Перший рівень. Синтаксична валідація за допомогою JSON Schema. Це перший і найшвидший етап валідації, що виконується при генерації структурного подання M_S у JSON. Його призначення – перевірка базової синтаксичної коректності та структури M_S. Для цього використовується файл schemas/structural_schema.json, який формалізує вимоги до JSON-документа, перевіряючи обов'язковість полів, формат ідентифікаторів за допомогою регулярних виразів, типи даних та унікальність елементів у масивах (uniqueItems).

Другий рівень. Імперативна інкрементальна валідація (в реальному часі). Це ядро інтерактивної роботи плагіна, що запускається при кожному збереженні файлу в IDE. Його призначення – перевірка семантичних та міжмодельних інваріантів, що потребують аналізу зв'язків між елементами. Алгоритм ValBeh(ΔM_B), реалізований у ValidationEngine, після зміни коду визначає множину порушених елементів (Affected Elements) за допомогою графа залежностей GMB, і перевірці підлягають лише вони. На цьому рівні перевіряються такі інваріанти, як BehaviorExists (чи є поведінка для динамічної операції), LifelineCorrespondence (чи відповідає лінія життя існуючому класу), SignatureMatches (чи збігаються сигнатури) та StartStateExists (чи є у діаграми станів початковий стан).

Третій рівень. Глибока верифікація через SAT-аналіз. Цей рівень використовується для перевірки складних глобальних властивостей моделі, які важко або неефективно перевіряти імперативними алгоритмами. Його призначення – це формальна верифікація транзитивних властивостей, насамперед ациклічності. Модуль validators/alloy_bridge.py виконує трансляцію частини моделі (наприклад, графа успадкування класів) у специфікацію мови Alloy та викликає SAT-вирішувач. Наприклад, для перевірки ациклічності успадкування генерується твердження assert NoCycles { no c: Class | c in c.^extends }. Alloy намагається знайти контрприклад (наприклад, цикл A → B → A). Через високу обчислювальну складність, цей аналіз запускається лише за прямою командою користувача або в CI/CD конвеєрі.

Також плагін надає CLI для інтеграції в автоматизовані процеси. В першу чергу, це використання через CLI для CI/CD:Основні команди, реалізовані в integration/cli.py, потенційно дозволяють вбудувати валідацію в будь-який CI/CD конвеєр (напр., GitHub Actions):

```
# Генерація моделей з вихідного коду
poetry run uml-consistency generate --project-root./project
# Повна валідація з JSON-виводом для аналізу скриптом
poetry run uml-consistency validate --format json --project-root./project
# Запуск глибокої SAT-перевірки глобальних інваріантів
poetry run uml-consistency sat-check –project-root./project
```

Опція --format json є важливою для автоматизації, оскільки дозволяє CI-скриптам легко парсити результати валідації та приймати рішення, наприклад, відмітити збірку як невдалу при наявності помилок.

Нижче наведено демонстраційний приклад, де розглядається сценарій, який демонструє перевагу плагіна: глибоку, семантичну узгодженість. Розробник виконує рефакторинг у Python-коді: перейменовує метод device.toggle_status() на device.set_active(). Однак він забуває оновити docstring-коментар з тегом @sequence, де цей метод викликається для опису діаграми послідовності: @sequence… -> device : toggle_status().

Плагін виконає наступні дії:

1. Інкрементальний парсер ASTParserVisitor фіксує зміну в UMLOperation (перейменування методу toggle_status).

2. Граф залежностей G_{MB} через *трасувальне* та *семантичне* ребра визначає, що від цієї операції залежить елемент UMLMessage у діаграмі послідовності, яка була згенерована з docstring-коментаря.

3. ValidationEngine запускає перевірку інваріанта SignatureMatches (або більш базового AnchorExists) для цього повідомлення.

В результаті плагін миттєво підсвічує рядок з @sequence в редакторі коду і видає інформативну помилку у консолі.

Додатково було проведено експериментальну оцінку продуктивності, для кількісної оцінки запропонованого підходу.

В якості тестового стенду виступає проект на Python середнього розміру, що містить 50 класів, 500 методів та близько 10 поведінкових діаграм, згенерованих з коду. Експерименти проводились на машині з процесором Intel Core i7 та 32 GB RAM.

Інструментами для порівняння виступали StarUML та Visual Paradigm, як представники двох основних підходів до моделювання.

Було проведено два сценарії тестування*:* повна валідація (вимірювання часу та пікового використання пам'яті для початкового аналізу всього проекту з нуля) та інкрементальна валідація (внесення впливової зміни в один «центральний» клас, наприклад, перейменування методу, від якого залежить багато інших елементів та вимірювання часу на повторну валідацію). Для аналогів, що не підтримують інкрементальний підхід, це означало повторну перевірку всієї моделі.

Результати експериментальної оцінки, представлені на рисунках 6 та 7, наочно демонструють практичні переваги запропонованого інкрементального підходу.

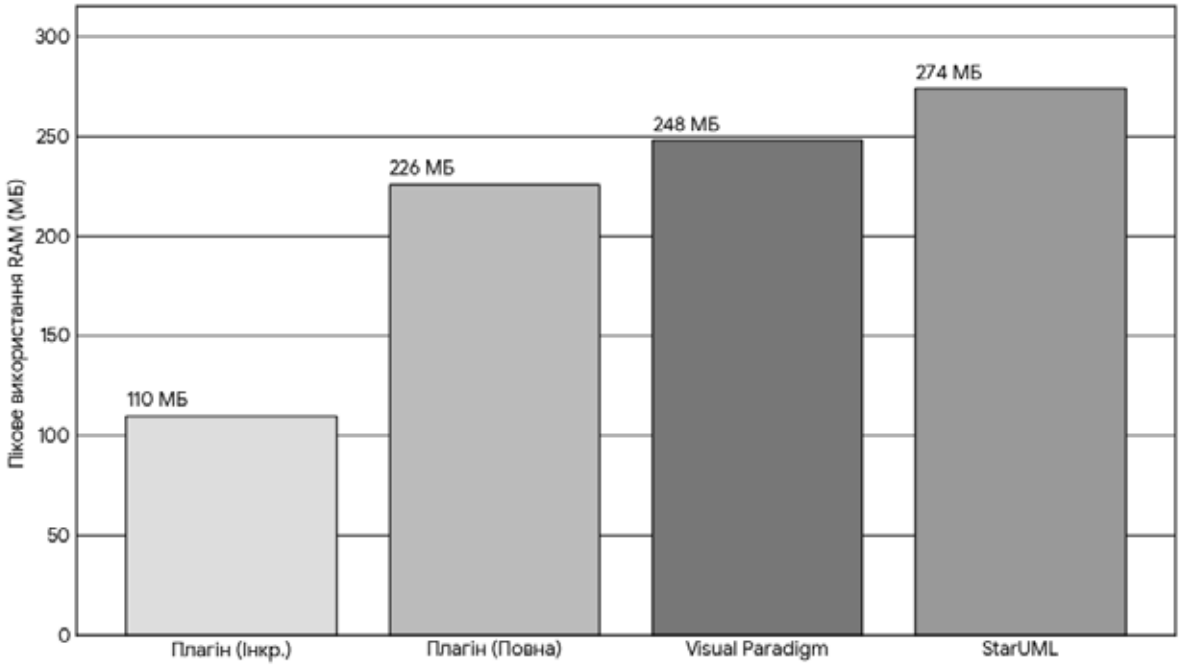

Рис. 6. Порівняння споживання пам'яті

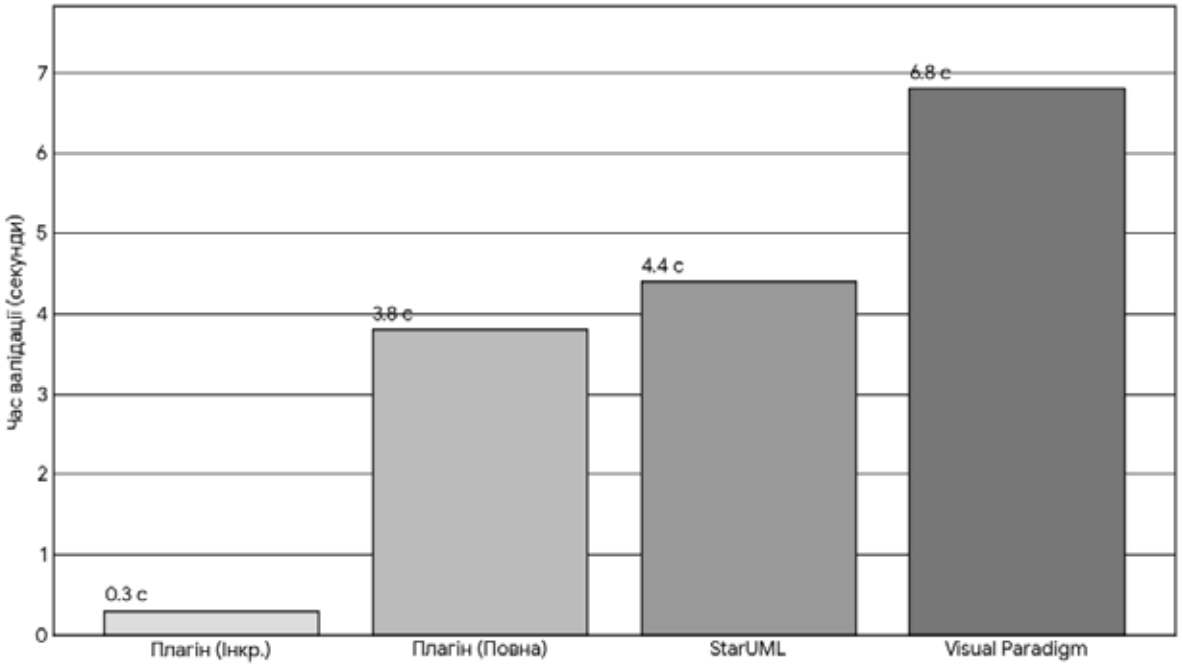

Рис. 7. Порівняння по витраченому часу

Аналіз результатів показує, що час інкрементальної валідації в плагіні є значно меншим, ніж час повної перевірки в аналогів. Це підтверджує теоретичну оцінку складності. У той час як конкуренти працюють за алгоритмом зі складністю O(N), де N – загальна кількість елементів моделі, наш інкрементальний підхід має складність O(k·d), де k – кількість змінених елементів (k≪N), а d – середній ступінь залежностей. Тому такий приріст продуктивності робить цей плагін єдиним інструментом з представлених, який є практично придатним для використання в режимі реального часу в IDE та для інтеграції у швидкі CI/CD конвеєри. Ефективне використання пам'яті під час

інкрементальної перевірки досягається за рахунок роботи лише з невеликою підмножиною моделі.

Висновки. У даній статті представлено архітектуру, практичну реалізацію та експериментальну оцінку плагіна, який є програмним артефактом, що втілює розроблену раніше теоретичну метамодель з двома поданнями. Проведене дослідження дозволяє зробити висновки, що запропонована архітектура, яка поєднує зберігання статичної структури в JSON та динамічної поведінки в XMI, є життєздатною та ефективною. Вона успішно вирішує фундаментальний конфлікт між гнучкістю, необхідною для сучасних ітеративних процесів розробки, та формальною строгістю, що вимагається для забезпечення надійності складних систем. Ключовий механізм зв'язку між поданнями довів свою спроможність забезпечувати надійне та швидке трасування між елементами моделі, що є основою для міжмодельної валідації. Також експериментально доведено, що інкрементальний підхід до валідації, заснований на графі залежностей G_{MB}, забезпечує значний приріст продуктивності порівняно з існуючими CASE-інструментами, які покладаються на повну перевірку моделі. Плагін успішно вирішує проблему виявлення прихованих семантичних неузгодженостей між вихідним кодом, його структурним поданням та поведінковими сценаріями – дефектів, які часто пропускаються аналогічними інструментами через їхню відірваність від коду.

В результаті, реалізація є комплексним рішенням, що пропонує новий підхід до розробки програмного забезпечення.

Список літератури:

1. Brambilla M., Cabot J., Wimmer M. Model-Driven Software Engineering in Practice: Second Edition. *Synthesis Lectures on Software Engineering*. 2017. Vol. 3, no. 1. P. 1–207. DOI: 10.2200/s00751ed2v01y201701swe004.

2. Cicchetti A., Ciccozzi F., Pierantonio A. Multi-view approaches for software and system modelling: a systematic literature review. *Software and Systems Modeling*. 2019. Vol. 18, no. 6. P. 3207–3233. DOI: 10.1007/s10270-018-00713-w.

3. Нікітченко М. І. Аналіз форматів збереження UML-моделей у сучасних CASE-засобах. *Технології та суспільство: взаємодія, вплив, трансформація : матеріали IV Міжнар. наук. конф. (Чернігів, 20 червня 2025 р.).* Чернігів, 2025. DOI: 10.62731/mcnd-20.06.2025.

4. Usman M., Zeshan F., Perwaiz A., Younas M. A Survey of Consistency Checking Techniques for UML Models. *Proceedings of the Second International Symposium on Advanced Software Engineering & Its Applications (ASEA), Hainan, China, 13–15 December 2008*. 2008. DOI: 10.1109/asea.2008.40.

5. Komleva N. O., Nikitchenko M. I. Method for Incremental Control of Consistency Between Structural and Behavioral Views of Software Architecture. *AAIT*. 2025. Vol. 8, no. 2. P. 162–177. DOI: 10.15276/aait.08.2025.11.

6. Комлева Н. О., Нікітченко М. І. Структурне подання UML-метамоделі у форматі JSON. *Інформатика та математичні методи в моделюванні*. 2025. Том 15, № 2. С. 205–217. DOI: 10.15276/imms.v15.no2.205.

7. Комлева Н. О., Нікітченко М. І. Поведінкове подання у форматі XMI в межах UML-метамоделі з двома поданнями. *Вісник Хмельницького національного університету*. Технічні науки. 2025. Том 5 (Подано до друку).

8. Altamimi T., Petriu D. Incremental Change Propagation from UML Software Models to LQN Performance Models. *Proceedings of the 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems, Austin, Texas, 17–22 September 2017.* 2017. P. 110–120. DOI: 10.1145/3121651.3121653.

Nikitchenko M.I. ARCHITECTURE AND IMPLEMENTATION OF A PLUGIN FOR INCREMENTAL VALIDATION OF UML METAMODELS WITH TWO VIEWS

The article addresses the issue of ensuring the consistency of UML models in the development of complex software systems, the relevance of which arises from the high cost of identifying architectural defects at later stages of the software life cycle. Current CASE tools for modeling offer either the flexibility of JSON format for static structures with low formalism or the strict of XMI format, which is challenging to integrate into daily development due to complexity. However, neither approach fully combines flexibility and formality, necessitating the development of a hybrid solution.

The purpose of this study is to present the architecture and practical implementation of a plugin integrating two formats for representing UML models–lightweight JSON for static structures and formally rigorous XMI for system dynamics description. A notable feature of the plugin is the use of a specialized UML profile to physically realize the correspondence relationship between these views, enabling rapid incremental validation through a dependency graph. The methodology combines JSON Schema syntax checking, semantic incremental validation, and deep formal verification of transitive properties using SAT analysis.

Experimental performance evaluation results demonstrate that the proposed plugin significantly outperforms existing analogs, as incremental validation reduces checking time compared to full model validation. This enables efficient integration into continuous integration and delivery pipelines, making the tool suitable for real-time use. The conclusions of the study highlight the potential of a hybrid UML model storage approach and outline future development directions, including support for additional programming languages and extended plugin functionalities.

Key words: *UML, model consistency, incremental validation, JSON Schema, XMI, dependency graph, SAT analysis.*

Дата надходження статті: 28.07.2025
Дата прийняття статті: 07.08.2025
Опубліковано: 27.10.2025