https://doi.org/10.31891/2307-5732-2025-357-42
УДК 004.45:004.021

КОМЛЕВА НАТАЛІЯ
Національний університет «Одеська політехніка»
https://orcid.org/0000-0001-9627-8530
e-mail: komleva@op.edu.ua
НІКІТЧЕНКО МАКСИМ
Національний університет «Одеська політехніка»
https://orcid.org/0009-0007-9560-7057
e-mail: maksym.nikitchenko@gmail.com

ПОВЕДІНКОВЕ ПОДАННЯ У ФОРМАТІ XMI В МЕЖАХ UML-МЕТАМОДЕЛІ З ДВОМА ПОДАННЯМИ

У статті запропоновано підхід до представлення поведінкового подання моделей мовою уніфікованого моделювання UML у форматі на основі міжплатформного обміну XML (XMI). Цей формат є складовою UML-метамоделі, що складається із структурного та поведінкового подань, де саме структурна інформація зберігається у форматі на основі об'єктної нотації JavaScript. Такий розподіл подання моделей забезпечує більшу модульність, кращу підтримку інструментальної перевірки та можливість інкрементального контролю узгодженості між структурними та поведінковими компонентами. Розроблено спеціалізований профіль для поведінкових моделей, що містить механізми трасування між поведінковими та структурними елементами, зокрема за допомогою атрибутів JsonRef, TriggerSource та інших розширень, які вказують на логічні зв'язки з об'єктами структури. Така реалізація дозволяє зберігати семантичну цілісність моделі навіть при фізичному поділі подання на дві частини.

У роботі визначено формалізовані множини елементів для опису структури та поведінки моделі, а також відображення між ними, що задаються у вигляді відношень відповідності. Для основних типів поведінкових діаграм (діаграм станів, діяльності і послідовностей) сформульовано систему логічних інваріантів, які виражають формальні вимоги до коректності моделей, зокрема щодо повноти умов, досяжності станів, відповідності викликів повідомлень та типізації параметрів. Запропоновано алгоритм локалізованої перевірки змін у поведінковому поданні моделі, що використовує побудову графа залежностей між елементами моделі для обмеження області повторної перевірки. В результаті це забезпечує значне зменшення обчислювальних витрат при редагуванні великих моделей у процесі розробки, зокрема в умовах безперервної інтеграції або послідовного рефакторингу.

Оцінено теоретичну складність роботи алгоритму, а також проаналізовано можливість його узагальнення на інші типи поведінкові діаграм. Представлені результати можуть бути застосовані в інструментах моделювання, орієнтованих на підтримку формальної перевірки моделей, модульного зберігання та автоматизованого контролю цілісності. Запропонований підхід формує основу для подальших досліджень у напрямі автоматизованої генерації поведінки з структурних специфікацій, а також двонапрямної трансформації традиційною XMI-формою та запропонованою UML-метамоделлю з метою забезпечення сумісності з зовнішніми UML-інструментами.

***Ключові слова**: метамодель, подання, інкрементальна валідація, консистентність, трасування,UML-профіль.*

KOMLEVA NATALIIA
NIKITCHENKO MAKSYM
Odesa Polytechnic National University

BEHAVIORAL VIEW IN XMI FORMAT WITHIN THE UML METAMODEL WITH TWO VIEWS

The article proposes an approach to representing behavioral models in the Unified Modeling Language (UML) in a format based on cross-platform XML exchange (XMI). This format is part of the UML metamodel, which consists of structural and behavioral views, where structural information is stored in a format based on JavaScript Object Notation. This division of model views provides greater modularity, better support for tool-based verification, and the ability to incrementally control consistency between structural and behavioral components. A specialized profile has been developed for behavioral models, containing mechanisms for tracing between behavioral and structural elements, in particular using JsonRef, TriggerSource, and other extensions that indicate logical relationships with structural objects. This implementation allows the semantic integrity of the model to be preserved even when the views are physically divided into two parts.

The paper defines formalized sets of elements for describing the structure and behavior of the model, as well as the mapping between them, which are specified in the form of correspondence relations. For the main types of behavioral diagrams (state, activity, and sequence diagrams), a system of logical invariants is formulated that express formal requirements for the correctness of models, in particular regarding the completeness of conditions, the reachability of states, the correspondence of message calls, and the typing of parameters. An algorithm for localized verification of changes in the behavioral view of a model is proposed, which uses the construction of a graph of dependencies between model elements to limit the scope of re-verification. As a result, this significantly reduces the computational cost of editing large models during development, particularly in continuous integration or sequential refactoring environments.

The theoretical complexity of the algorithm is evaluated, and the possibility of generalizing it to other types of behavioral diagrams is analyzed. The presented results can be applied in modeling tools focused on supporting formal model verification, modular storage, and automated integrity control. The proposed approach forms the basis for further research in the direction of automated behavior generation from structural specifications, as well as bidirectional transformation between the traditional XMI form and the proposed UML metamodel in order to ensure compatibility with external UML tools.

***Keywords**: metamodel, view, incremental validation, consistency, tracing, UML profile.*

Стаття надійшла до редакції / Received 02.08.2025
Прийнята до друку / Accepted 25.08.2025

Постановка проблеми у загальному вигляді та її зв'язок із важливими науковими чи практичними завданнями

У сучасній інженерії програмного забезпечення та складних систем широкого поширення набули підходи, що базуються на моделюванні, зокрема з використанням Unified Modeling Language (UML). Для опису різних аспектів системи (структурних, поведінкових, інформаційних) часто застосовують окремі, але взаємопов'язані подання. Такий підхід підвищує модульність та спрощує супровід моделей, проте породжує важливу науково-практичну проблему підтримання їхньої узгодженості.

Актуальним напрямом є розробка метамоделей, де структурну частину моделі представлено у форматі JSON, орієнтованому на швидкість обробки та економію пам'яті, а поведінкову – у стандартизованому форматі XML Metadata Interchange (XMI). Однак такий розподіл створює виклик: стандарт XMI орієнтований на повну UML-модель, і при відокремленні структурної інформації втрачається контекст, необхідний для валідації поведінкових діаграм. Інструментальні засоби не мають вбудованих механізмів для трасування елементів поведінки в XMI до їхніх джерел у зовнішньому JSON-файлі, що унеможливлює автоматизований контроль цілісності.

В результаті виникає актуальна задача розробки нового підходу до представлення та перевірки поведінкових моделей у форматі XMI, який би враховував залежності від зовнішнього структурного подання. Практична реалізація такого підходу потребує створення спеціалізованого UML-профілю для збереження трасувальних зв'язків, формалізації інваріантів коректності та розробки ефективних алгоритмів інкрементальної валідації, що дозволить інтегрувати перевірку узгодженості в автоматизовані цикли розробки.

Огляд пов'язаних досліджень та стандартів

Model-driven engineering (MDE) є провідною парадигмою для проектування складних програмних систем, де UML виступає як індустріальний стандарт для візуалізації, специфікації та документування архітектури [1]. Офіційна специфікація UML від Object Management Group (OMG) визначає її семантику та нотацію. Але для повного опису системи часто потрібні численні подання, що відображають її різноманітні аспекти, в тому числі структурні та поведінкові. Це неминуче породжує проблему забезпечення узгодженості між ними, як показано в систематичному огляді літератури [2, 3].

Проблема підтримки консистентності є доволі важливою в MDE. Дослідники визначили сотні потенційних правил узгодженості між різними діаграмами UML [4], а також розробили керівні принципи для управління якістю моделей [5]. Існують комплексні підходи, такі як Vitruvius [6], що пропонують фреймворки для синхронізації різноманітних подань. Основою для формалізації правил узгодженості є мова об'єктних обмежень (OCL), що стала, по суті, стандартом для визначення інваріантів у моделях UML [7].

Класична перевірка всієї моделі після кожної зміни є неефективною для великих систем. Тому значна увага приділяється методам інкрементальної валідації, які аналізують лише змінені елементи та їх залежності, суттєво скорочуючи обчислювальні витрати [8]. Для гарантування коректності використовуються строгі формальні методи. Серед них виділяються підходи на основі потрійних графових граматик для двонапрямної трансформації та синхронізації моделей [9], аналіз за допомогою SAT-вирішувачів, наприклад, Alloy, для перевірки структурних діаграм [10], та використання мереж Петрі для верифікації поведінкових моделей [11]. Інші формальні підходи також довели свою ефективність у виявленні суперечностей на ранніх етапах [12].

Технічною основою для обміну моделями, як вже було сказано вище, є XMI, а для розширення семантики UML під конкретні завдання, таких як додавання механізмів трасування, застосовуються UML-профілі. При роботі з дуже великими моделями, що зберігаються в різних форматах, виникають проблеми продуктивності, які вирішуються за допомогою спеціалізованих фреймворків, таких як NeoEMF, що забезпечує ефективне зберігання моделей у мультиформатних середовищах [13].

У контексті практичного застосування, окремі інструменти UML-моделювання (зокрема, StarUML, Modelio, Enterprise Architect) надають можливість експорту або обробки моделей у JSON або XMI, однак не забезпечують інтеграції обох форматів на рівні однієї метамоделі [14]. Деякі промислові фреймворки застосовують JSON як легку форму зберігання структурної частини (наприклад, класів, атрибутів, зв'язків), доповнюючи її XMI-документами для поведінкових сценаріїв або симуляцій. Дослідницькі публікації, що розглядають метамоделі з багатьма поданнями, як правило, зосереджуються на трансформаціях між форматами. Однак вони не формалізують повноцінну схему верифікації узгодженості, через що досі відсутні усталені стандарти для валідації моделей з комбінованою структурою.

Проведений аналіз показує, що жоден з існуючих стандартів, профілів чи інструментальних засобів не забезпечує повної підтримки поведінкового подання UML у форматі XMI при умові, що структурне подання моделі представлена у форматі JSON. Зокрема:

− відсутні засоби трасування поведінкових елементів до структурних у зовнішньому форматі;

− не підтримується декларативне збереження зовнішніх типів і атрибутів без втрати сумісності;

− не реалізовані інкрементальні перевірки змін у поведінковому поданні з урахуванням зовнішніх залежностей.

Ці обмеження створюють прогалину, яка потребує формального методологічного заповнення. Запропонована в цій статті методика є спробою подолати цю прогалину шляхом побудови поведінкового профілю UML для XMI, доповненого інваріантами та механізмом інкрементальної валідації на основі трасування до JSON-структури.

Формулювання цілей статті

Мета роботи полягає у розробці формального підходу до представлення та валідації поведінкових діаграм UML у форматі XMI в межах метамоделі з двома поданнями, де структурне подання реалізована у форматі JSON, що дозволяє підвищити модульність моделей та ефективність автоматизованого контролю їхньої цілісності.

Методологія. Для досягнення поставленої мети використано апарат метамоделювання UML, теорію формальних мов та логіки для визначення інваріантів коректності, а також теорію графів для побудови моделі залежностей, що лежить в основі алгоритму інкрементальної валідації.

Об'єктом дослідження є процес забезпечення семантичної та структурної узгодженості UML-моделей з декількома поданнями.

Предметом дослідження є метод представлення та інкрементальної валідації поведінкового подання UML-метамоделі у форматі XMI з урахуванням її зв'язків зі структурним поданням, представленою у форматі JSON.

Наукова новизна роботи полягає у формульовані системи формальних інваріантів для поведінкових діаграм (станів, діяльності, послідовностей), які враховують зовнішні структурні залежності та забезпечують перевірку коректності моделі в умовах двох подань.

Виклад основного матеріалу

Розглянемо UML-метамодель, що складається з двох взаємопов'язаних подань:

- M_S – множина елементів структурного подання моделі, представлених у форматі JSON;
- M_B – множина елементів поведінкового подання моделі, представлених у форматі XMI;
- μ: $M_B \rightarrow M_S \cup M^2_S$ – часткове відображення, що задає відповідність між поведінковими елементами та їх структурними джерелами.

Відображення μ є не тотальним, оскільки не всі елементи поведінки потребують відповідника у структурі (наприклад, допоміжні переходи або повідомлення), однак усі структурно-залежні поведінкові об'єкти (наприклад, lifeline, trigger, targetState) повинні бути трасованими до відповідного елемента в M_S.

Формально, для кожного елемента $b \in M_B$ існує нуль або одна структурна відповідність $\mu(b)$, яка може бути об'єктом $s \in M_S$ для простого відображення або впорядкованою парою $(s_1, s_2) \in M^2_S$ для зв'язків між елементами. Це дозволяє описувати як *монолінійні залежності (наприклад, «подія → клас»), так і бінарні* (наприклад, «повідомлення → (відправник, отримувач)»).

У межах цього дослідження розглядається обмежена, але достатньо репрезентативна підмножина поведінкових діаграм UML: діаграми станів, діаграми діяльності та діаграми послідовностей. Ці діаграми охоплюють найбільш поширені сценарії специфікації поведінки в інженерії програмних систем і добре формалізовані в специфікації UML. Інші типи поведінкових діаграм, зокрема діаграми синхронізацій або огляду взаємодій, на даному етапі не розглядаються через відсутність потреби у складній часовій логіці або розподіленому управлінні.

Поведінкове подання M_B зберігається у форматі XMI. При цьому він повинен мати характерну структурну відповідність стандарту UML 2.5, тобто всі елементи M_B повинні бути коректно типізовані відповідно до відповідних UML-метакласів. Самі данні повинні бути провалідовані за схемою або DTD, що відповідає UML-моделі, та не містити синтаксичних або семантичних порушень. Для збереження специфічних зв'язків з M_S допускається використання UML-профілю з додатковими тегами та стереотипами (наприклад, jsonId, payloadType), які не порушують сумісності зі стандартними засобами XMI-обробки. І, нарешті, файл з M_B повинен зберігати повну поведінкову інформацію навіть у разі відсутності структурної частини M_S в XMI, тобто бути частково самодостатнім.

Вхідними артефактами для вирішуваної задачі є:

- структурна модель у форматі JSON (M_S), яка містить опис класів, атрибутів, типів, асоціацій і зберігається відповідно до визначеної JSON Schema або еквівалентної метамоделі;
- поведінкова модель у форматі XMI (M_B), яка включає підтримувані діаграми з використанням UML-профілю та специфічних тегів;
- відображення μ (явно або неявно), що зв'язує елементи M_B з M_S та зберігається або у вигляді тегів (напр., jsonRef="customer_42"), або у формі окремої таблиці трасування.

На підставі вище описаного формалізуємо задачу перевірки поведінкового подання у форматі XMI у межах UML-метамоделі як задачу перевірки трикратної відповідності:

1) збереження семантики при поділі на JSON/XMI. Поведінкова модель M_B повинна містити всю необхідну інформацію для виконання моделі, за умови що зв'язок із M_S (через μ) дозволяє повністю реконструювати семантику поведінки. Формально: $\forall b \in M_B$, якщо type(b) вимагає контексту $s \in M_S$, то $\mu(b)=s$ і s існує;
2) забезпечення коректності (валідації). Існує система інваріантів I_B, які повинні бути виконані для всіх $b \in M_B$. Ці інваріанти формалізують синтаксичні, семантичні та структурні вимоги

до поведінкових моделей з урахуванням зовнішніх структурних посилань. Формально: $\forall b \in M_B$, $I_B(b)=true$;

3) збереження відповідності між M_S і M_B. Відображення μ повинно бути повним для всіх структурно-залежних елементів M_B та бути консистентним з поточним станом M_S. Для кожного $b \in M_B$, що вимагає зв'язку: $b \models Consistent(\mu(b), M_S)$, де $Consistent(s, M_S)$ – предикат, що перевіряє існування s у поточній версії M_S та відповідність типу, імені, тощо.

В результаті, задача валідації поведінкового подання зводиться до перевірки виконання інваріантів M_B у контексті відображення μ на поточну структуру M_S.

Для забезпечення можливості інтегрованого представлення поведінкових діаграм у форматі XMI в межах UML-метамоделі з двома поданнями, де структурне подання зберігається у форматі JSON, було розроблено спеціалізований UML-профіль. Він дозволяє зберігати додаткову інформацію, необхідну для зв'язку поведінкових елементів з об'єктами структурного подання, а також для збереження семантичних атрибутів, які інакше були б втрачені у стандартному XMI-кодуванні.

Механізм профілювання UML передбачає створення надбудов над метамоделлю UML шляхом введення нових стереотипів, тегів та обмежень, шо, в сукупності, дозволяє розширювати семантику UML без модифікації основної метамоделі, зберігаючи повну сумісність із XMI-форматом та інструментами, які підтримують профілі UML. У запропонованому підході UML-профіль використовується як контейнер для включення додаткової інформації про структурну прив'язку, зокрема – до елементів M_S, які не зберігаються безпосередньо в XMI. Ці прив'язки реалізуються через теговані значення, що забезпечують трасування елементів M_B до M_S без втрати сумісності (табл. 1).

Таблиця 1

Стереотипи та теги UML-профілю для трасування між поведінковим та структурним поданнями

Стереотип	**Базовий елемент UML**	**Теги (тип)**	**Призначення / приклад використання**
«JsonRef»	State, Action, Lifeline, Trigger	jsonId: String	Ідентифікатор елемента з M_S, до якого прив'язано поведінковий елемент.
«PayloadType»	Message, Signal	type: String	Тип переданого значення або об'єкта, узгоджений з типом атрибута в M_S.
«TriggerSource»	Transition	sourceId: String	Посилання на структурне джерело події (наприклад, метод чи подія в класі).
«SemanticTag»	Bci	contextInfo: String	Додатковий семантичний опис, який не може бути збережений у стандартних UML-атрибутах.
«StateCategory»	State	kind: Enumeration {simple, composite, final}	Класифікація станів для експрес-валідації правил завершеності.
«GuardExpr»	Transition	expression: String	Збереження тексту логічної умови переходу (якщо зберігається, напр., у JSON-застосунку).
«EventType»	Trigger	category: Enumeration {signal, call, time, change}	Дозволяє швидко виконувати перевірку віднесення тригерів до структурних подій.
«EndpointRef»	Lifeline	endpointId: String	Посилання на компонент або мікросервіс, описаний у структурному поданні.
«ExceptionType»	Message	exception: Boolean	Маркування повідомлень типу «fault» для подальшої валідації сценаріїв винятків.
«TimingInfo»	State, Action	deadline: String, period: String	(Не обов'язково) Зберігання часових характеристик, сумісне з MARTE.
«JsonPath»	будь-який	path: String	Повний JSON Path до відповідного елемента M_S, якщо одного jsonId недостатньо.

Перераховані стереотипи дозволяють зберігати зв'язки з структурним поданням та забезпечити їх подальше відновлення при аналізі моделі. У XMI-документі інформація з профілю кодується у вигляді спеціальних елементів-розширень <uml:Model>, які містять посилання на відповідні стереотипи, при цьому не порушують синтаксичної валідності XMI і можуть бути опрацьовані спеціалізованими інструментами.

Наприклад, зв’язок стану з відповідним класом у JSON може бути представлено наступним чином:

```
<packagedElement xmi:type="uml:State" xmi:id="state1" name="Processing">
  <xmi:Extension extender="UMLProfile">
    <JsonRef jsonId="Class#Customer"/>
  </xmi:Extension>
</packagedElement>
```

Щоб UML-інструмент міг коректно обробляти запропонований профіль, він повинен підтримувати стандартну механіку UML-профілів (тобто читання/запис стереотипів, тегів, обмежень), дозволяти розширення XMI-файлів з елементами <xmi:Extension>, забезпечувати можливість імпорту/експорту профілю разом з модельними даними та надавати API або розширюваний механізм для доступу до тегованих значень. Серед інструментів, які вже частково відповідають цим вимогам, можна відзначити Enterprise Architect, MagicDraw/Cameo, Papyrus і Modelio.

Запропонований профіль спроектовано таким чином, щоб не порушувати специфікації стандарту UML 2.5.1 та формату XMI, де усі елементи збережено в рамках дозволених XMI-розширень. Крім того, профіль є несуперечним до fUML і Alf, оскільки не вводить нових елементів виконуваної семантики, а лише додає трасувальні та описові теги. Профіль також сумісний із використанням інших стандартних UML-профілів (наприклад, SysML або MARTE), оскільки уникає перевизначення типів або властивостей, що вже визначені в них. Це дозволяє інтегрувати описану поведінкову модель в розширені інженерні середовища з підтримкою багатьох спеціалізованих профілів.

Окремо необхідно задати систему формальних інваріантів, які гарантують синтаксичну, семантичну та структурну цілісність поведінки. Це потрібно для того, щоб забезпечити коректність поведінкового подання, особливо у випадку розподіленої моделі із відокремленим структурним контекстом. Перш за все потрібно сформувати основні правила для діаграми станів.

Як відомо, діаграма станів є одним із базових типів поведінкових моделей у UML, що формалізує допустимі переходи між станами об'єкта залежно від подій і умов. Наведемо основні інваріанти для підмножини елементів діаграми станів: *StateMachine*, *State*, *Transition*, *Trigger*, *Guard*. Формалізація виконана у вигляді логічних предикатів (переважно першого порядку). Ці інваріанти можуть застосовуватись як для повної перевірки (offline), так і для інкрементальної валідації локальних змін.

Інваріант **StartStateExists** гарантує, що будь-яка діаграма станів містить однозначно визначений початковий елемент, який ініціює виконання поведінки:

$$\forall s_m \in StateMachine, \exists! s \in s_m.substates : isInitial(s).$$

Інваріант **TransitionDefined** гарантує, що кожен перехід повинен мати визначені початковий і цільовий стан, які існують у межах однієї діаграми.

$$\forall t \in Transition, \exists s_{src}, s_{dst} \in States : t.source = s_{src} \wedge t.target = s_{dst}.$$

Інваріант **GuardCompleteness** вимагає, щоб кожен умовний перехід мав явно задану логічну умову, яка визначає допустимість переходу під час виконання.

$$\forall t \in Transition, requiresGuard(t) \Rightarrow \neg isEmpty(t.guard).$$

Інваріант **TransitionDeterminism** гарантує детермінованість поведінки, тобто з одного й того самого стану не може існувати кілька переходів із однаковими умовами та подією.

$$\forall s \in States, \nexists t_1, t_2 \in s.outgoing, t_1 \neq t_2 : t_2.trigger = t_2.trigger \wedge t_1.guard = t_2.guard.$$

Інваріант **StateReachability** гарантує, що кожен стан повинен бути досяжним з початкового стану через скінченну послідовність переходів.

$$\forall s \in States, \exists \pi = \langle t_1, \dots, t_k \rangle : t_1.source = s_0, t_k.target = s.$$

Інваріант **TriggerCorrespondence** гарантує, що якщо тригер посилається на структурний елемент (подію, метод), то він має бути трасований через тег jsonId до відповідного елемента в M_S. У контексті профілю це означає, що перевіряється наявність стереотипу «JsonRef» і валідність його параметра.

$$\forall trig \in Trigger, \exists s \in M_S : \mu(trig) = s.$$

Діаграма діяльності у UML визначає поведінку системи як орієнтований граф, що описує послідовність дій (Action), логіку розгалужень, обробку подій і передавання об’єктів. Центральними елементами такої діаграми є вузли діяльності, які об'єднуються потоками керування (ControlFlow) та об’єктними потоками (ObjectFlow). Для забезпечення коректності поведінкової моделі, особливо в умовах її узгодження зі структурним поданням, необхідно також виконувати перевірку низки формальних інваріантів.

Однією з ключових конструкцій діаграми діяльності є вузол типу Pin. Цей елемент слугує для передавання даних між діями та моделює точки входу (InputPin) або виходу (OutputPin) об’єктів відповідного типу, тому Pin є семантичним посередником між структурою моделі (яка визначає типи об’єктів) та поведінковим описом, у якому ці об’єкти циркулюють під час виконання.

У рамках перевірки валідності такої моделі можна виділити наступні основні інваріанти.

Інваріант **ActionConnected** гарантує, що кожен вузол типу Action повинен бути включений щонайменше в один потік – або як джерело, або як приймач. Інакше кажучи, дії, які не мають жодного зв'язку з іншими елементами діаграми, розглядаються як некоректні, оскільки вони не можуть бути досягнуті або активовані під час виконання:

$$\forall a \in A_N,\ isAction(a) \Rightarrow (\exists f \in F_L : tgt(f) = a \vee \exists f \in F_L : src(f) = a).$$

Інваріант **FlowValid** вимагає, що кожен потік повинен поєднувати вузли, типи яких сумісні за семантикою UML. Зокрема, *ControlFlow* з'єднує лише вузли керування (наприклад, InitialNode, Action, DecisionNode), тоді як *ObjectFlow* має зв'язувати вузли, що репрезентують об'єкти (тобто Pin або ObjectNode) і типи яких узгоджені:

$$\forall f \in F_L, isControlFlow(f) \Rightarrow C(src(f)) \wedge C(tgt(f)), isObjectFlow(f) \Rightarrow O(src(f), tgt(f)),$$

де C(·) – перевірка, чи вузол є контрольним,

O(x,y) – перевірка узгодженості типів і наявності принаймні одного Pin-вузла.

Інваріант **InputMatchesType** гарантує, що для кожного InputPin повинен бути визначений тип даних, який узгоджується з відповідним елементом у структурному поданні моделі. Зв'язок між InputPin та елементом структури здійснюється за допомогою тегу jsonId, що вказує на відповідний атрибут або параметр у структурному поданні. Перевірка цього інваріанта гарантує збереження типового узгодження між структурною та поведінковою частинами моделі.

$$\forall p \in InputPin,\ \exists s \in M_S : \mu(p) = s \wedge type(p) = type(s).$$

Інваріант **DecisionGuardDefined** вимагає, що кожен вузол розгалуження (DecisionNode) повинен мати принаймні дві вихідні дуги, і кожна з них повинна мати визначену логічну умову. Це забезпечує однозначність вибору шляху під час виконання діяльності:

$$\forall d \in A_N, isDecisionNode(d) \Rightarrow (|\{f \mid src(f) = d\}| \geq 2 \wedge \forall f\ (src(f) = d \Rightarrow \neg isEmpty(guard(f))))$$

Інваріант **FinalNodeReachable** гарантує, що модель вважається завершеною лише тоді, коли щонайменше один вузол завершення (ActivityFinalNode або FlowFinalNode) досяжний від початкового вузла за скінченною послідовністю потоків. Цей інваріант забезпечує завершуваність процесу, змодельованого активністю:

$$\exists fn \in A_N : isFinalNode(fn) \wedge Reachable(init, fn)$$

Діаграми послідовностей у UML призначені для опису сценаріїв взаємодії об'єктів у часі. Основними елементами є лінії життя (Lifeline), які представляють об'єкти, та повідомлення (Message), що передаються між ними. У рамках поточної метамоделі, цілісність і семантична правильність таких діаграм повинні перевірятися відповідно до формальних інваріантів, які забезпечують узгодження, правильну типізацію та порядок викликів.

Інваріант **LifelineCorrespondence** забезпечує структурну відповідність моделі учасників сценарію. Кожна лінія життя (lifeline) повинна відповідати певному елементу у структурному поданні, такому як клас або екземпляр, що бере участь у поведінці. Зв'язок задається через тег jsonId у стереотипі «JsonRef»:

$$\forall l \in Lifeline,\ \exists s \in M_S : \mu(l) = s$$

Інваріант **MessageConsistency** перевіряє семантичну коректність комунікації: відповідність операцій, імен, типів. Для кожного повідомлення (Message) має бути вказано ім'я операції, а також типи параметрів повинні бути узгоджені з операцією відповідного класу в структурному поданні. Також потрібно забезпечити, що отримувач повідомлення (receiver) підтримує відповідний метод:

$$\forall m \in Message,\ \exists c \in M_S : \mu(receiver(m)) = c \wedge \exists op \in Operations(c): name(op) = name(m) \wedge params(op) = params(m)$$

Інваріант **OrderingPreserved** верифікує логіку виконання: повідомлення не можуть змінюватися місцями всупереч причинно-наслідковим зв'язкам. Порядок повідомлень, як визначено у поведінковому поданні, має відповідати допустимому виклику методів у системі. Зокрема, кожне наступне повідомлення не повинно починатись раніше за завершення попереднього, якщо між ними є логічна залежність. Це стосується переважно синхронних повідомлень, де вимагається завершення виклику до переходу до наступного:

$$\forall (m_1, m_2) \in Message^2 : depends(m_2, m_1) \Rightarrow end(m_1) \prec start(m_2),$$

де depends(m_2, m_1) – предикат залежності (може бути визначений, наприклад, через ланцюг відповідей), а $\prec$ – відношення порядку за часом або позицією в XMI.

Для забезпечення цілісності UML-метамоделі з двома поданнями необхідно сформулювати механізм відображення між цими частинами. Цей механізм слугує основою для перевірки інваріантів, які включають посилання на структурні елементи моделі. Основою такого узгодження виступає відношення трасування $\mu \subseteq M_B \times M_S$, що реалізується через спеціалізовані стереотипи, теговані значення та ідентифікатори посилання.

Для забезпечення явного зв'язку між об'єктами в M_B та відповідними елементами в M_S використовується стереотип «JsonRef», що додається до таких елементів, як Trigger, InputPin, Lifeline, Message, тощо. У межах цього стереотипу визначено тег jsonId, який містить унікальний ідентифікатор відповідного елемента в структурному поданні.

Цей підхід дозволяє реалізувати функцію μ як часткову відповідність:

$$\mu(eM_B) = eM_S,$$

де $e_{mb} \in M_B$,

$e_{ms} \in M_S$,

e_{mb}.jsonId = ems.id

В результаті, будь-яке звернення до структурної інформації з боку інваріанта (наприклад, перевірка типу, перевірка наявності атрибута або операції) здійснюється через значення поля jsonId у поведінковій моделі.

Також визначимо типову схему інтеграції. У діаграмах діяльності InputPin вказує на атрибут класу; перевірка типу InputMatchesType відбувається через jsonId. У діаграмах послідовностей Lifeline вказує на клас/екземпляр; перевірка LifelineCorrespondence та MessageConsistency здійснюється через цю відповідність. У діаграмах станів Trigger вказує на атрибут/подію; перевірка TriggerCorrespondence базується на наявності правильного jsonId.

Узагальнене правило відповідності виглядає наступним чином. Нехай $P \subseteq M_B$ – множина поведінкових елементів, для яких визначено зв'язок з M_S, тоді має виконуватись:

$$\forall p \in P, \exists! s \in M_S: \mu(p) = s,$$

тобто для кожного поведінкового елемента, який потребує узгодження, існує єдиний структурний відповідник.

Інваріанти, які посилаються на M_S (наприклад, InputMatchesType, MessageConsistency), передбачають обхід JSON-структури на основі значень jsonId, тому перевірка таких інваріантів потребує доступу до обох частин моделі, а також реалізації механізму семантичного узгодження між форматами XMI і JSON.

У процесі моделювання поведінки програмної системи в інструментальному середовищі розробник здійснює поступові зміни у поведінковому поданні UML-діаграм. При цьому повна повторна перевірка всіх інваріантів моделі після кожної модифікації є неефективною з обчислювального погляду та знижує інтерактивність середовища. Тому виникає потреба у **інкрементальній валідації**, яка перевіряє лише ті частини поведінкової моделі, які потенційно могли втратити коректність через зміну.

Інкрементальна валідація передбачає формулювання задачі наступним чином: для кожної зміни в M_B, визначити підмножину інваріантів та відповідних елементів, які потребують повторної перевірки, а також залежності з боку M_S, якщо такі існують.

З формального погляду, нехай:

- ΔM_B – множина змінених елементів поведінкового подання;
- Inv – множина всіх інваріантів поведінкового подання;
- Inv(e) – підмножина інваріантів, що застосовуються до елемента $e \in M_B$;
- μ: $M_B \nrightarrow M_S$ – відображення елементів M_B на відповідники в M_S.

Тоді множина інваріантів, які підлягають перевірці після зміни, визначається як:

$$Inv^{\Delta} = U_{e \in \Delta MB} Inv(e) \cup Inv(\mu(e))$$

Тобто до перевірки включаються інваріанти, пов'язані зі зміненими поведінковими елементами, та інваріанти, пов'язані з елементами M_S, які з ними узгоджені.

Крім цього, для інваріантів з неявною залежністю (наприклад, StateReachability, OrderingPreserved), може знадобитися транзитивне розширення області перевірки за графом залежностей. Тому інструментальна підтримка повинна реалізовувати механізми виявлення залежностей між об'єктами (наприклад, depends(f_2, f_1) для переходів чи повідомлень) та побудови мінімального супермножини локального контексту перевірки. При цьому локалізована перевірка дозволяє зменшити час валідації, зберегти інтерактивність редактора моделі та забезпечити безперервну перевірку коректності навіть у процесі редагування.

У контексті інкрементальної валідації поведінкового подання M_B, важливо формалізувати типи змін, що виникають у процесі моделювання. Кожна зміна ΔM_B характеризується типом операції: add, delete або update. Додавання описує появу нових елементів у моделі (таких як стани, дії, лінії життя чи переходи) і потребує перевірки інваріантів, пов'язаних із типом нового елемента, його контекстом та, у разі наявності трасувального посилання, відповідністю з елементами M_S. Видалення об'єктів з моделі може призвести до порушення структурної цілісності або логіки виконання, зокрема до втрати зв'язків між елементами, недосяжності кінцевого стану, порушення узгодженості з M_S. У цьому випадку активується перевірка як безпосередньо суміжних елементів, так і інваріантів, що охоплюють ширший контекст. Оновлення передбачає зміну властивостей об'єктів без їх фізичного вилучення, наприклад, редагування виразів логічних умов переходів, параметрів повідомлень або значень тегів (jsonId). Такі зміни можуть вплинути на виконання інваріантів, чутливих до семантики чи трасувальної відповідності.

В результаті, множина ΔM_B є вихідною основою для побудови цільової підмножини інваріантів, що потребують повторної перевірки, адже її структура дозволяє ефективно локалізувати ділянки поведінкової моделі, які потенційно втратили коректність у результаті редагування, і застосувати оптимізовані механізми валідації.

Однією з ключових умов інкрементальної валідації є можливість обмежити перевірку лише тими частинами моделі, які прямо або опосередковано залежать від змінених елементів. Для цього вводиться поняття графа залежностей поведінкового подання, що описує відношення впливу між об'єктами у M_B.

Нехай поведінкове подання M_B складається з множини елементів:

$$E_{MB} = \{e_1, e_2, \dots, e_n\},$$

де кожен e_i є окремим об'єктом у XMI-файлі (наприклад, State, Transition, Action, Message). Визначимо орієнтований граф залежностей:

$$G_{MB} = (E_{MB}, R),$$

де вершини відповідають об'єктам поведінкового подання, а дуга $(e_i, e_j) \in R$ означає, що елемент e_j залежить від e_i, тобто зміна e_i може призвести до порушення інваріантів, пов'язаних з e_j.

У графі залежностей виділяють три логічні різновиди ребер. Структурні залежності відображають чисто синтаксичні зв'язки між елементами XMI: перехід пов'язаний із вихідним і цільовим станами, потік керування з'єднує дію з вузлом розгалуження, а повідомлення «прикріплюється» до відповідної лінії життя. Семантичні залежності фіксують власне поведінкову логіку: тригер спрацьовує від події, що надходить із повідомлення; вхідний Pin дії посилається на конкретний параметр або тип даних; логічна умова переходу використовується для перевірки значення атрибутів. Нарешті, трасувальні залежності підтримують узгодженість зі структурним поданням M_S: стереотипи JsonRef чи TriggerSource через атрибут jsonId вказують на відповідні класи, атрибути або операції, визначені у JSON-структурі, гарантуючи цілісність між двома поданнями моделі.

Граф залежностей формується автоматично під час початкового розбору XMI-файлу або оновлюється інкрементально при зміні елементів. В рамках парсингу здійснюється:

- витягування всіх об'єктів $e \in E_{MB}$;
- ідентифікація атрибутів, які містять посилання на інші об'єкти (IDREF, href, jsonId);
- створення орієнтованих дуг (e_i, e_j) за кожним відношенням залежності.

Після кожної зміни $\delta \in \Delta M_B$ обчислюється множина елементів, на які вона впливає, за допомогою транзитивного замикання:

$$Affected(\delta) = \{e' \in E_{MB} \vee \text{існуєшлях} \delta \to e' \text{у} G_{MB}\}.$$

Далі для кожного $e' \in$ Affected(δ) визначається відповідна підмножина інваріантів Inv(e′), які підлягають повторній перевірці.

У підсумку, повна множина інваріантів, які слід перевірити після змін, становить:

$$Inv^{\Delta} = \cup_{\delta \in \Delta MB} \cup_{e' \in Affected(\delta)} Inv(e')$$

Такий підхід дозволяє значно зменшити обсяг валідації. Замість перевірки всієї моделі, система аналізує ті елементи, які дійсно могли зазнати порушення консистентності. Це особливо важливо для складних діаграм з десятками чи сотнями об'єктів, де повна перевірка є надто витратною за часом. Граф залежностей виконує роль інтелектуального фільтра і забезпечує контекстно-обмежену, швидкодіючу валідацію в реальному часі.

Розглянемо спрощений приклад поведінкового подання M_B, а також побудований для нього граф залежностей між основними елементами подання (рис. 1). Цей граф містить шість об'єктів: два стани S1 і S2, перехід T1 між ними, тригер TR1, що активує цей перехід, повідомлення MSG1, з яким пов'язаний тригер, а також дію A1, яка виконується у стані S2.

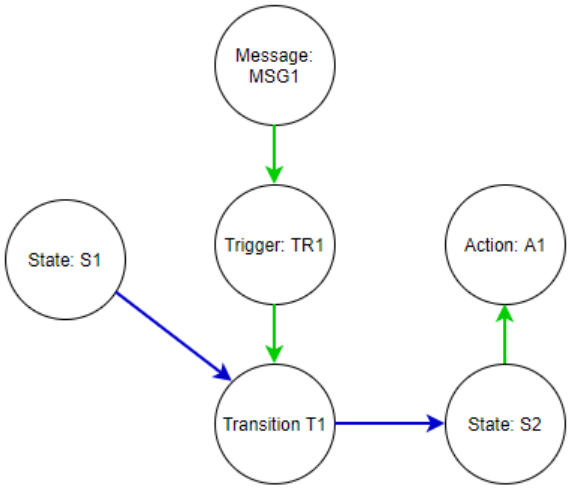

Рис. 1. Граф залежностей між елементами поведінкової моделі UML

Між цими об'єктами встановлено кілька типів зв'язків. Структурні залежності (синій колір) представлені, зокрема, зв'язками між станами і переходом: перехід T1 має вихідний стан S1 та цільовий

стан S2. Ці зв'язки відображають синтаксичну структуру діаграми. Далі, семантичні залежності (зелений колір) фіксують поведінкову логіку моделі. Повідомлення MSG1 активує тригер TR1, який у свою чергу прикріплений до переходу T1. Оскільки цей перехід веде до стану S2, у якому виконується дія A1, між відповідними елементами також встановлюються залежності, що враховуються при валідації семантичних інваріантів.

Тому, при зміні одного з елементів – наприклад, повідомлення MSG1 – система за допомогою графа визначає всі об'єкти, які залежать від нього: TR1, T1, S2, A1. Це дозволяє сконцентрувати перевірку лише на тих інваріантах, які можуть бути порушені в результаті такої зміни, що значно скорочує обсяг валідації й підвищує її ефективність.

Додатково отримав подальший розвиток алгоритм інкрементальної перевірки поведінкових інваріантів ValBeh(ΔM_B), який реалізує новий підхід до валідації UML-метамоделей зі структурним та поведінковим поданнями у форматах JSON і XMI.

Новизна алгоритму ValBeh(ΔM_B) полягає в наступному:

- вперше запропоновано підхід до інкрементальної валідації поведінкового подання UML у комбінованому використанні форматів JSON та XMI;
- побудовано формальний граф залежностей із розмежуванням ребер на структурні, семантичні та трасувальні;
- підтримано трасування між поданнями M_S і M_B через відображення $\mu \subseteq M_B \times (M_S \cup M_S \times M_S)$;
- забезпечено індексацію інваріантів за типами об'єктів, що дозволяє точно локалізувати перевірку після змін.

Алгоритм приймає на вхід множину змінених елементів:

$$\Delta M_B = \{b_1, b_2, \dots, b_k\},$$

і визначає для кожного $b_i \in \Delta M_B$ множину зачеплених елементів через обхід графа залежностей G_{MB}:

$$Affected = \cup_{b \in \Delta MB}\, Affected(b).$$

Для кожного елемента e ∈ Affected визначається множина інваріантів Inv(e), пов'язаних із його типом, з яких формується загальний список для перевірки:

$$InvToCheck = \cup_{e \in Affected}\, Inv(e).$$

Кожен інваріант з цієї множини перевіряється окремо, після чого формується звіт про виявлені порушення.

Складність інкрементальної валідації визначається насамперед числом змінених елементів $k=|\Delta M_B|$ та глибиною їх залежностей у графі G_{MB}. Для кожного зміненого елемента виконується обхід його локального оточення в графі, який включає всі елементи, пов'язані через структурні, семантичні або трасувальні ребра.

Якщо позначити середню кількість зачеплених елементів як d, то загальна оцінка верхньої межі часу роботи алгоритму має вигляд:

$$O(k \cdot d),$$

де $k=|\Delta M_B|$, а d – верхня межа глибини або ширини залежностей (у разі обмеженої транзитивності).

На практиці $d \ll |M_B|$, що забезпечує суттєве скорочення витрат порівняно з повною валідацією, яка має складність $O(|M_B|)$.

Висновки

У роботі представлено формалізований підхід до подання поведінкових аспектів UML-моделі у форматі XMI в межах UML-метамоделі із двома поданнями, де структурне подання представлено у форматі JSON. Запропонована методологія дозволяє забезпечити семантичну узгодженість між структурними та поведінковими поданнями за допомогою спеціальних трасувальних механізмів, таких як JsonRef і TriggerSource, що підтримують зв'язок між елементами XMI та JSON. Для ключових типів поведінкових діаграм (станів, діяльності і послідовностей) визначено формальні інваріанти, які контролюють як синтаксичну правильність, так і відповідність структурному контексту.

Одним із значущих результатів дослідження є подальший розвиток алгоритму ValBeh(ΔM_B) для інкрементальної перевірки поведінкових інваріантів. Алгоритм базується на побудові графа залежностей між елементами поведінки, який класифікує зв'язки як структурні, семантичні або трасувальні. Це дозволяє локалізувати валідацію лише до тих частин моделі, які дійсно зазнали змін або залежать від змінених елементів, що, у свою чергу, забезпечує значне зниження обчислювальних витрат. Формально обґрунтовано, що в разі обмеженої глибини залежностей складність перевірки зменшується з $O(|M_B|)$ до $O(k \cdot d)$, де k – кількість змінених елементів, а d – середня кількість пов'язаних з ними об'єктів.

Науковий внесок цієї роботи полягає у формалізації інваріантів поведінкового подання, побудові універсального графа залежностей для локалізації перевірок та розробці алгоритмічної схеми інкрементальної валідації. З практичної точки зору, результати можуть бути безпосередньо інтегровані в UML-сумісні інструменти з підтримкою профілювання та трасування, що відкриває перспективи застосування в CI/CD-процесах, системах модельної еволюції та автоматизованого аналізу архітектур.

Разом із тим, запропонований підхід має певні обмеження: його ефективність залежить від наявності повного трасувального зв'язку між поведінкою та структурою, а також від здатності інструментального середовища інтерпретувати інваріанти в заданому форматі. Окрім того, не всі типи поведінкових UML-діаграм наразі охоплено.

Перспективним напрямом подальших досліджень є розробка двонапрямного механізму трансформації, що забезпечить семантично еквівалентне перетворення між канонічним, монолітним представленням UML-моделей у форматі XMI та запропонованою структурою з двома поданнями, що забезпечить сумісність із зовнішніми інструментами та підтримку імпорту/експорту моделей.

Література

1. Brambilla M., Cabot J., Wimmer M. Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering. 2012. Vol. 1, no. 1. P. 1–182. URL: https://doi.org/10.1007/978-3-031-02549-5.

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. URL: https://doi.org/10.1007/s10270-018-00713-w.

3. 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. URL: https://doi.org/10.15276/aait.08.2025.11.

4. Lucas F. J., Molina F., Toval A. A systematic review of UML model consistency management. Information and Software Technology. 2009. Vol. 51, no. 12. P. 1631–1645. URL: https://doi.org/10.1016/j.infsof.2009.04.009.

5. UML models consistency management: Guidelines for software quality manager / R. S. Bashir et al. International Journal of Information Management. 2016. Vol. 36, no. 6. P. 883–899. URL: https://doi.org/10.1016/j.ijinfomgt.2016.05.024.

6. Enabling consistency in view-based system development – The Vitruvius approach / H. Klare et al. Journal of Systems and Software. 2021. Vol. 171. P. 110815. URL: https://doi.org/10.1016/j.jss.2020.110815.

7. The object constraint language: Getting your models ready for MDA / ed. by K. A. G. Boston, MA : Addison-Wesley, 2003. 236 p.

8. Clarisó R., A. González† C., Cabot J. Incremental Verification of UML/OCL Models. The Journal of Object Technology. 2020. Vol. 19, no. 3. P. 3:1. URL: https://doi.org/10.5381/jot.2020.19.3.a7.

9. Leblebici E., Anjorin A., Schürr A. Inter-model Consistency Checking Using Triple Graph Grammars and Linear Optimization Techniques. Fundamental Approaches to Software Engineering. Berlin, Heidelberg, 2017. P. 191–207. URL: https://doi.org/10.1007/978-3-662-54494-5_11.

10. Maoz S., Ringert J. O., Rumpe B. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. Model Driven Engineering Languages and Systems. Berlin, Heidelberg, 2011. P. 592–607. URL: https://doi.org/10.1007/978-3-642-24485-8_44.

11. Thierry-Mieg Y., Hillah L.-M. UML behavioral consistency checking using instantiable Petri nets. Innovations in Systems and Software Engineering. 2008. Vol. 4, no. 3. P. 293–300. URL: https://doi.org/10.1007/s11334-008-0065-0.

12. A Formal Approach for Consistency Management in UML Model / H. Wen et al. International Journal of Software Engineering and Knowledge Engineering. 2023. URL: https://doi.org/10.1142/s0218194023500134.

13. NeoEMF: A multi-database model persistence framework for very large models / G. Daniel et al. Science of Computer Programming. 2017. Vol. 149. P. 9–14. URL: https://doi.org/10.1016/j.scico.2017.08.002.

14. 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. URL: https://doi.org/10.62731/mcnd-20.06.2025.

References

1. Brambilla M., Cabot J., Wimmer M. Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering. 2012. Vol. 1, no. 1. P. 1–182. URL: https://doi.org/10.1007/978-3-031-02549-5.

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. URL: https://doi.org/10.1007/s10270-018-00713-w.

3. 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. URL: https://doi.org/10.15276/aait.08.2025.11.

4. Lucas F. J., Molina F., Toval A. A systematic review of UML model consistency management. Information and Software Technology. 2009. Vol. 51, no. 12. P. 1631–1645. URL: https://doi.org/10.1016/j.infsof.2009.04.009.

5. UML models consistency management: Guidelines for software quality manager / R. S. Bashir et al. International Journal of Information Management. 2016. Vol. 36, no. 6. P. 883–899. URL: https://doi.org/10.1016/j.ijinfomgt.2016.05.024.

6. Enabling consistency in view-based system development – The Vitruvius approach / H. Klare et al. Journal of Systems and Software. 2021. Vol. 171. P. 110815. URL: https://doi.org/10.1016/j.jss.2020.110815.

7. The object constraint language: Getting your models ready for MDA / ed. by K. A. G. Boston, MA : Addison-Wesley, 2003. 236 p..

8. Clarisó R., A. González C., Cabot J. Incremental Verification of UML/OCL Models. The Journal of Object Technology. 2020. Vol. 19, no. 3. P. 3:1. URL: https://doi.org/10.5381/jot.2020.19.3.a7.

9. Leblebici E., Anjorin A., Schürr A. Inter-model Consistency Checking Using Triple Graph Grammars and Linear Optimization Techniques. Fundamental Approaches to Software Engineering. Berlin, Heidelberg, 2017. P. 191–207. URL: https://doi.org/10.1007/978-3-662-54494-5_11.

10. Maoz S., Ringert J. O., Rumpe B. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. Model Driven Engineering Languages and Systems. Berlin, Heidelberg, 2011. P. 592–607. URL: https://doi.org/10.1007/978-3-642-24485-8_44.

11. Thierry-Mieg Y., Hillah L.-M. UML behavioral consistency checking using instantiable Petri nets. Innovations in Systems and Software Engineering. 2008. Vol. 4, no. 3. P. 293–300. URL: https://doi.org/10.1007/s11334-008-0065-0.

12. A Formal Approach for Consistency Management in UML Model / H. Wen et al. International Journal of Software Engineering and Knowledge Engineering. 2023. URL: https://doi.org/10.1142/s0218194023500134.

13. NeoEMF: A multi-database model persistence framework for very large models / G. Daniel et al. Science of Computer Programming. 2017. Vol. 149. P. 9–14. URL: https://doi.org/10.1016/j.scico.2017.08.002.

14. Nikitchenko M. I. Analysis of UML model storage formats in modern CASE tools. Technology and society: interaction, influence, transformation: materials from the IV International Scientific Conference (Chernihiv, June 20, 2025). Chernihiv, 2025. URL: https://doi.org/10.62731/mcnd-20.06.2025.