DOI 10.15276/imms.v15.no2.205
УДК 004.42: 004.021

Informatics and Mathematical Methods in Simulation
Vol.15 (2025), No. 2, pp. 205-217

СТРУКТУРНЕ ПОДАННЯ UML-МЕТАМОДЕЛІ У ФОРМАТІ JSON

Н.О. Комлева, М.І. Нікітченко

Національний університет «Одеська політехніка»
1, Шевченка пр., м. Одеса, 65044, Україна
Emails: komleva@op.edu.ua, maksym.nikitchenko@gmail.com

В роботі розглянуто проблему формального представлення моделей UML (Unified Modeling Language) у відкритому форматі, придатному для перевірки, трансформації та інтеграції в інженерні процеси розробки програмного забезпечення. Як концептуальну основу дослідження використано UML-метамодель, що включає два взаємопов'язані подання: структурне (визначає об'єкти та зв'язки моделі) та поведінкове (описує активності, стани й послідовності). Така метамодель слугує базою для інкрементального контролю консистентності програмної архітектури. У даній роботі особливу увагу приділено структурному представленню метамоделі UML, який є базовим для відображення архітектури системи у вигляді взаємопов'язаних типів об'єктів. Для цього представлення реалізовано формалізоване представлення ключових метасутностей UML, таких як класи, атрибути, асоціації, пакети, компоненти та об'єкти, у форматі JSON (JavaScript Object Notation). Обране подання базується на застосуванні стандарту JSON Schema, що дозволяє чітко описувати допустимі структури, типи даних, іменовані властивості та правила валідації. У результаті побудовано повноцінну множину типів і відношень, які визначають дозволену конфігурацію UML-моделі як структурованого набору об'єктів із чітко визначеними зв'язками між ними. Для підтримки синтаксичної і семантичної цілісності цієї моделі було сформульовано систему формалізованих інваріантів, яка фіксує критичні обмеження, зокрема, вимоги до ідентифікаторів, коректного вкладення елементів, суворої типізації атрибутів, а також відсутності циклічних залежностей у композиційних і успадкованих структурах. Це закладає основу для уніфікованого представлення структурних моделей та подальшої автоматизованої перевірки їх коректності. Частина правил реалізована засобами самої JSON-схеми, а для складніших передбачено виконання через зовнішні механізми перевірки. Запропонований підхід дозволяє забезпечити чітку відповідність між абстрактними об'єктами UML та їх представленням у відкритому форматі, що, в перспективі, відкриває можливість уніфікованої інтеграції з іншими засобами моделювання, CI/CD та інструментами генерації коду. Дослідження зосереджено на формалізмі, прозорості та узгодженості з UML-специфікацією, які є необхідними для підтримки інкрементального контролю консистентності із забезпеченням сумісності між складовими метамоделі.

Ключові слова: моделювання, UML, метамодель, JSON Schema, валідація, інваріанти.

Вступ. UML (Unified Modeling Language) є загальновизнаним стандартом для візуального моделювання програмних систем. У типовому випадку UML-модель охоплює кілька взаємодоповнювальних подань, серед яких найбільш поширеними є структурні – діаграми класів, компонентів, об'єктів, пакетів, що відображають статичну архітектуру системи – та поведінкові, які охоплюють динамічні аспекти функціонування через діаграми активностей, станів, послідовностей, комунікацій тощо. Додатково до них можуть використовуватись фізичні (розгортання), вимог (використання) та інші подання, що формують повну модель програмної системи.

У даній роботі розглянуто проблему формального представлення моделей UML у відкритому форматі, придатному для перевірки, трансформації та інтеграції в інженерні процеси розробки програмного забезпечення. В основі дослідження – метамодель UML, що складається зі структурного подання для фіксації об'єктів і зв'язків та поведінкового

для опису активностей і послідовностей [1], яка забезпечує інкрементальний контроль консистентності моделей.

У межах даної статті зосереджено увагу на структурному представлені, який є критично важливим для подальшої верифікації та узгодженості моделі системи. Особливу увагу приділено реалізації цього подання у форматі JSON (JavaScript Object Notation), який розглядається як альтернатива традиційному XMI (XML Metadata Interchange). Незважаючи на те, що XMI є офіційним стандартом OMG (Object Management Group), він часто виявляється надмірно складним, громіздким і малопридатним до інкрементальної обробки в CI/CD-середовищах.

Натомість JSON є легковагим текстовим форматом, що набув широкого використання в сучасних інструментах моделювання (наприклад, StarUML використовує .mdj, Modelio – .uml.json). Проте, відсутність єдиного формалізованого стандарту для JSON-подання UML обмежує можливість гарантованої перевірки моделей, перешкоджає автоматизації трансформацій та ускладнює вбудовування у сучасні інженерні процеси [2]. Саме тому виникає необхідність у створенні структурного подання метамоделі, здатної формально описати допустимі об’єкти, зв’язки та обмеження UML-моделі у вигляді строго визначених типів та правил, виражених у термінах JSON Schema.

Огляд існуючих рішень. Значну увагу дослідники приділяють забезпеченню консистентності UML/OCL (Object Constraint Language) моделей та перевірці інваріантів при інкрементальних змінах. У роботі [3] запропоновано метод логічної абстракції для інкрементальної верифікації, що дозволяє ефективно перевіряти часткові оновлення без повної повторної валідації. Систематизований огляд технік обрізання моделей та методів надання зворотного зв’язку з метою підвищення ефективності перевірки діаграм класів UML/OCL наведено у [4].

Паралельно з розвитком UML-моделювання поширюється використання формату JSON, який забезпечує сумісність із сучасними веб-орієнтованими технологіями. Офіційні специфікації JSON Schema для опису структури й обмежень JSON-документів, придатних для валідації UML-моделей, наведено в [5; 6]. Відповідність UML-структур формату JSON та правила їх кодування згідно з рекомендаціями OGC представлені у [7]. Проблеми формалізації та складності валідації JSON-схем, а також методи спрощення таких перевірок, детально розглядаються у [8; 9].

Окрім традиційних підходів до перевірки UML-моделей з використанням OCL, актуальними є також онтологічні методи. У [10] досліджено використання reasoner-інструментів для перевірки узгодженості класів, об’єктів та діаграм станів UML, що відкриває перспективи застосування семантичних технологій у моделюванні. Узагальнений огляд формальних і практичних методів забезпечення консистентності UML-класів подано в [11]. На практичному рівні ведуться спроби трансформувати JSON-схеми у UML-структури для задач зворотної інженерії, прикладом чого є реалізований конвертер JSONSchema-to-UML [12]. Актуальність теми підтверджується також документацією StarUML [13] та Modelio [14], де розкрито структуру відповідних форматів (.mdj, .uml.json), які зберігають UML-моделі на основі JSON.

У напрямі онтологічної трансформації UML-класів варто відзначити роботу [15], в якій запропоновано метод формалізованого перетворення UML-моделей з використанням онтологічних обмежень для подальшої перевірки їхньої коректності. Перспективи колаборативного UML-моделювання у хмарних середовищах та вимоги до взаємодії інструментів обговорюються в [16], де підкреслюється важливість формальних підходів для підтримки інтеграції. Нарешті, дослідження [17] акцентує на формалізації чотиришарової архітектури метамоделювання, що є основою для подальшого розвитку інструментів валідації UML-моделей з урахуванням рівнів абстракції.

Таким чином, аналіз літератури підтверджує, що поєднання формального представлення UML-моделей у форматі JSON з механізмами автоматизованої валідації є

перспективним напрямом, що потребує подальшого розвитку як на методологічному, так і на інструментальному рівнях.

Мета дослідження. Метою даного дослідження є побудова формалізованого структурного подання UML- метамоделі у форматі JSON, яка забезпечує однозначне відображення ключових метасутностей на елементи JSON Schema. Таке подання поєднує формальну строгість опису UML-структури з гнучкістю JSON-середовища, що створює підґрунтя для розробки інструментів автоматизованої валідації, трансформації та генерації коду на основі UML-моделей у відкритому форматі.

Формалізоване структурне подання UML-метамоделі у форматі JSON. У межах цього дослідження запропоновано структурне подання UML-метамоделі у форматі JSON, що задається у вигляді пари:

$$M_S^{JSON} = < T, R >, \quad (1)$$

де T – множина допустимих типів структурних метасутностей, відображених у форматі JSON; R – множина структурних відношень між об'єктами, які також представлені як обмеження на JSON-структури.

У контексті підтримуваних діаграм UML множина типів T включає:

T = {UMLClass, UMLAttribute, UMLOperation, UMLAssociation, UMLGeneralization, UMLEnumeration, UMLPackage, UMLComponent, UMLObject}.

Множина відношень R задає ключові структурні залежності між об’єктами:

- hasAttribute ⊆ UMLClass × UMLAttribute;
- hasOperation ⊆ UMLClass × UMLOperation;
- typedBy : UMLAttribute → (UMLClass ∪ Primitive);
- generalizes ⊆ UMLClass × UMLClass;
- associates ⊆ UMLAssociation × UMLClass.

Таким чином, кожна діаграма UML (класи, компоненти, об’єкти, пакети) моделюється як часткове відображення цієї загальної структури M_S^{JSON}, з відповідними обмеженнями на типи, відношення та інваріанти, що накладаються через JSON Schema.

Кожен об'єкт o∈T у структурному поданні описується як структурований JSON-об’єкт, що має фіксований набір атрибутів, такі як ідентифікатор (поле id), ім’я (поле name), вкладені елементи або посилання (наприклад, attributes, associations, dependencies), типова специфікація (наприклад, type, superClassId, interface). Крім того, на множину елементів застосовуються валідаційні інваріанти $\Sigma=\{\sigma_1,\ldots,\sigma_n\}$, які формалізують правила структурної коректності моделі.

Вимоги до структурного подання UML у JSON. Для формальної обробки UML-моделей у JSON-форматі необхідно задати чіткі вимоги, які гарантують коректність, уніфікованість, інтероперабельність та розширюваність моделі.

Основні вимоги до структурного подання, що охоплює статичні аспекти (класи, атрибути, зв’язки, пакети):

- Метасутності: підтримка базових елементів UML, таких як Class, Attribute, Operation, Association, Package, Enumeration, Interface. Описуються у JSON через вкладені об'єкти;
- Ідентифікація об’єктів: кожен елемент повинен мати унікальний id, що дозволяє здійснювати посилання між елементами;
- Цілісність структури: заборонено цикли композиції та успадкування; граф структури має бути ациклічним. Визначається на етапі валідації графу моделі;
- Читабельність та логічність: імена полів, типів та вкладених елементів мають бути послідовними, стандартизованими, зрозумілими. Наприклад, name, visibility, type;
- Розширюваність (стереотипи): передбачена підтримка метаінформації: стереотипів, тегів, профільних атрибутів (якщо потрібно);

– Формат зв’язків: всі взаємозв’язки реалізуються через посилання на id елементів, а не через вкладеність;
– Сумісність із JSON Schema: структура повинна відповідати специфікаціям JSON Schema Draft 2019–09 або новішим, для можливості формальної валідації моделі.

Формалізація вимог до структурного подання UML-моделі у форматі JSON забезпечує створення представлення, яке є інтерпретованим, уніфікованим та яке можна легко перевірити. Така структура придатна до автоматизованої обробки, зокрема, валідації, трансформацій між поданнями, інкрементальної синхронізації та інтеграції в інженерні конвеєри розробки (CI/CD).

Проектування уніфікованої JSON-схеми. Загальна структура уніфікованого JSON-документа, що репрезентує структурне подання UML-моделі, має відповідати принципам формальної коректності, інтерпретованості та розширюваності. Основними елементами є наступні.

Кореневий об’єкт (root) є стартовою точкою документа. Він містить загальні метадані (версію, ідентифікатор, тип моделі), а також вказівки на колекції сутностей. Наприклад:

```
{
  "schemaVersion": "1.0",
  "modelType": "UMLStructure",
  "collections": { "$ref": "#/definitions/collections" }
}
```

Колекції (collections) групують елементи UML-моделі за категоріями: класи, асоціації, пакети тощо. Це дозволяє зберігати логічну організацію моделі. Кожна колекція є масивом об'єктів певного типу:

```
"collections": {
  "classes": [ { "$ref": "#/definitions/Class/U1" }, ... ],
  "packages": [ { "$ref": "#/definitions/Package/P1" } ]
}
```

Механізм посилань ($ref) базується на специфікації JSON Schema та OpenAPI, і забезпечує уніфіковане посилання на внутрішні чи зовнішні елементи моделі. Це дозволяє реалізувати повторне використання, уникати дублювання, та забезпечити чіткість структури.

Опис ключових типів у JSON Schema / OAS. Для формалізації структурного подання UML-моделі у форматі JSON на основі специфікацій JSON Schema та OpenAPI Specification (OAS) визначено ключові типи, які відповідають метасутностям UML: класам, атрибутам, операціям, асоціаціям, пакетам і перелікам (табл. 1). Кожен тип представлено як об'єкт із чітко визначеними полями, обов’язковими атрибутами й валідаційними обмеженнями. Головним типом є UMLClass, що описує сутності домену, їх атрибути (UMLAttribute), операції (UMLOperation), асоціації (UMLAssociation), пакети (UMLPackage) та переліки значень (UMLEnumeration).

Таблиця 1.

Узагальнена таблиця типів у JSON Schema / OAS

Тип	Призначення	Ключові поля	Особливості валідації
UMLClass	Представлення класу UML	id, name, attributes, operations, superClass, isAbstract, stereotypes	Перевірка унікальності id, допустимості типів
UMLAttribute	Опис атрибутів класу	name, type, visibility, defaultValue, multiplicity	Обов’язкові поля, перевірка діапазону значень
UMLOperation	Представлення методів класу	name, parameters, returnType, visibility	Валідація параметрів як масиву об’єктів

Тип	Призначення	Ключові поля	Особливості валідації
UMLAssociation	Визначення асоціації між класами	end1, end2, type, multiplicity, roleName	Обов'язковість двох кінців, правильність типів
UMLPackage	Ієрархічна організація елементів моделі	name, elements, stereotypes	Перевірка коректності імен та вкладеності
UMLEnumeration	Опис перелікових типів	name, literals	literals – перелік рядків із перевіркою унікальності

Усі типи підтримують механізм $ref для модульності та повторного використання, а також розширюються через додаткові поля (additionalProperties). Запропонована структура типів у JSON забезпечує формалізоване подання UML-сутностей, пряме проведення валідації моделей, їх застосування в інструментах моделювання та інтеграцію у процеси CI/CD.

Валідаційні правила подання JSON-схеми. Запропонована JSON-схема підтримує основні типи UML-діаграм із формалізацією відповідних метасутностей та валідаційних обмежень. Основний фокус спрямовано на діаграму класів, де підтримано метасутності Class, Attribute, Operation, Association, Generalization, Package й Enumeration. Діаграма компонентів представлена частково, за допомогою спеціалізованих Component-об'єктів або класів зі стереотипами для забезпечення узгодженості моделі. Діаграма об'єктів поки підтримується опосередковано: екземпляри описуються як об'єкти класів, але повна валідація conformant-інстанціювання поки не реалізована. Діаграма пакетів інтегрована через сутність Package, що дозволяє ієрархічно організовувати структуру моделі.

Запропонований модульний розподіл валідаційних правил має кілька переваг: він дозволяє поступово розширювати модель без ризику порушення стабільних частин, забезпечує цільову перевірку діаграм, знижує складність обчислень та гарантує прозорий зв'язок із UML-специфікацією. Для автоматизованої валідації UML-моделей у JSON-схемі формалізовано структурні обмеження через набір предикатів $\Sigma=\{\sigma_1, \sigma_2,\ldots,\sigma_n\}$, кожен із яких представляє окреме правило коректності моделі.

Згідно з запропонованим підходом, нижче наведено розширене формалізоване подання JSON-схеми UML-діаграми класів із описом ключових сутностей, основних валідаційних вимог і математичної інтерпретації; інваріанти валідації подано в табл. 2.

Визначимо для діаграми класів множину позначень: C – множина класів (UMLClass); A – множина атрибутів (UMLAttribute); O – множина операцій (UMLOperation); R – множина асоціацій (UMLAssociation); G – множина узагальнень (Generalization); ID – множина допустимих ідентифікаторів; означення бінарного відношення узагальнення $Gen \subseteq C \times C$, де $(c, p) \in Gen$ означає «клас c безпосередньо успадковує клас p»; означення множини предків $Ancestors(c) = \{ p \in C \mid (c, p) \in Gen \}$.

Таблиця 2.

Інваріанти валідації класів

№	Назва інваріанта	Формалізація / опис
σ_1	Валідність ідентифікатора класу	$\forall c \in C$: $id(c) \in ID \wedge name(c) \in \Sigma^+$
σ_2	Унікальність імен в межах пакета	$\forall c_1, c_2 \in C$: $package(c_1) = package(c_2) \wedge name(c_1) = name(c_2) \Rightarrow c_1 = c_2$
σ_3	Відсутність циклів узагальнення	$\neg\exists c \in C$: $c \in Ancestors^+(c)$, де $Ancestors(c) = \{p \in C \mid (c,p) \in Gen\}$
σ_4	Унікальність імен атрибутів у класі	$\forall a_1, a_2 \in A$: $class(a_1) = class(a_2) \wedge name(a_1) = name(a_2) \Rightarrow a_1 = a_2$

№	Назва інваріанта	Формалізація / опис
σ_5	Асоціації не є петлями	$\forall r \in R: end_1(r) \neq end_2(r)$
σ_6	Валідність типів атрибутів	$\forall a \in A: type(a) \in CU\{String, Integer, Boolean, Float\}$
σ_7	Унікальність асоціацій між парами	$\forall r_1, r_2 \in R: (end_1(r_1), end_2(r_1)) = (end_1(r_2), end_2(r_2)) \Rightarrow r_1 = r_2$

Наведемо JSON-фрагмент у якості прикладу базової схеми для опису UML-моделі у форматі JSON, зосередженої на двох ключових складових: класах (classes) та асоціаціях (associations). Така структура дозволяє формалізувати діаграму класів з мінімальним, але достатнім набором обмежень для забезпечення її валідації.

```
{
  "type": "object",
  "required": ["classes", "associations"],
  "properties": {
    "classes": {
      "type": "array",
      "items": {
        "type": "object",
        "required": ["id", "name", "attributes"],
        "properties": {
          "id": { "type": "string", "pattern": "^[A-Za-z_][A-Za-z0-
9_]*$" },
          "name": { "type": "string" },
          "attributes": {
            "type": "array",
            "items": {
              "type": "object",
              "required": ["name", "type"],
              "properties": {
                "name": { "type": "string" },
                "type": { "type": "string" }
              }
            },
            "uniqueItems": true
          }
        }
      },
      "uniqueItems": true
    },
    "associations": {
      "type": "array",
      "items": {
        "type": "object",
        "required": ["end1", "end2"],
        "properties": {
          "end1": { "type": "string" },
          "end2": { "type": "string", "not": { "const": { "$data":
"1/end1" } } }
        }
      }
    }
  }
}
```

У межах кореневого об’єкта визначено два обов’язкові поля: масив класів та масив асоціацій. Кожен клас повинен мати унікальний ідентифікатор (id), назву (name) і набір атрибутів (attributes). Ідентифікатор задається рядком, який відповідає регулярному виразу, що унеможливлює використання некоректних символів. Атрибути класу також представлені масивом об’єктів, кожен з яких повинен містити ім’я та тип. Обидва ці поля є обов’язковими, що забезпечує структурну завершеність опису класу.

Крім того, в описі атрибутів та класів використано вимогу uniqueItems, яка забороняє дублювання елементів у межах масиву. Це важливо для збереження унікальності імен класів або атрибутів.

Асоціації описуються як об'єкти з двома кінцями (end1 і end2), які вказують на пов'язані класи. Для запобігання петлям передбачено спеціальне правило, яке забороняє, щоб асоціація зв'язувала клас сам із собою.

Можна вважати, що така схема закладає основу для мінімально достатньої перевірки синтаксичної коректності UML-діаграми класів у JSON-поданні. Водночас, вона може бути розширена додатковими обмеженнями, такими як валідація типів, перевірка існування цільових об'єктів для асоціацій або підтримка успадкування та композиції.

У JSON-схемі, що описує *діаграму пакетів*, основним об'єктом виступає колекція packages. Кожен пакет визначається як окремий JSON-об'єкт, який обов'язково має поля id, name та elements. Формальні вимоги до структури пакета подані у табл. 3.

Для формалізації моделі пакету введемо наступні позначення: P – скінченна множина пакетів; C – множина класів (може бути розширена компонентами, переліками тощо); $E = P \cup C$ – сукупність структурних елементів, які може містити пакет; $id : E \to ID$ – функція, що відображає елемент у його унікальний ідентифікатор; $ID \subseteq \Sigma^+$; $name : E \to \Sigma+$ – функція іменування (людиночитабельності – текстове позначення об'єкта, яке призначене не для машинної обробки, а для сприйняття людиною); $contains \subseteq P \times E$ – бінарне відношення включення (пакет містить елемент).

Для виявлення циклічних залежностей у структурі пакетів використовується транзитивне замикання відношення contains, позначене як $contains^+$, що означає: якщо існує послідовність включень $p_1 \to p_2 \to \cdots \to p_k$, то $(p_1, p_k) \in contains^+$. Це замикання дозволяє перевірити, чи не містить пакет сам себе – прямо або опосередковано.

Таблиця 3.

Інваріанти схеми пакету

№	Назва інваріанта	Формалізація / опис
σ_1	Унікальність ідентифікаторів	$\forall e1,e2 \in E: id(e_1) = id(e_2) \Rightarrow e1 = e2$
σ_2	Унікальність імен у пакеті	$\forall p \in P, \forall e1,e2: (p,e1),(p,e2) \in contains, name(e1) = name(e2) \Rightarrow e1=e2$
σ_3	Ациклічність вкладеності	$\forall p \in P: (p,p) \notin contains^+$
σ_4	Коректність домену відношення	$\forall (p,e) \in contains: p \in P, e \in E$
σ_5	Відсутність дублювання підлеглих елементів	$\forall p \in P, \forall e: (p,e) \in contains \Rightarrow \nexists e' \neq e: (p,e') \in contains \wedge id(e) = id(e')$

Ідентифікатор id задається у вигляді рядка, який повинен відповідати регулярному виразу (наприклад, ^[A-Za-z_][A-Za-z0-9_\\-\\.]*$), що забезпечує як синтаксичну коректність, так і можливість глобальної унікальності в межах моделі (інваріант σ_1).

Поле name призначене для відображення у графічному інтерфейсі й не обов'язково повинне бути унікальним глобально, але має бути унікальним у межах одного пакета (σ_2). Усе вмістиме пакета визначається у полі elements, яке є масивом, що може включати об'єкти типів Class, Package та за потреби – інших сутностей (наприклад, Enumeration, Component). Типізація реалізується через конструкцію oneOf, що дозволяє зберігати гнучкість структури, але гарантує, що всі елементи пакета належать до допустимої множини (σ_4).

Щоб запобігти повторному включенню одного і того ж елемента, до масиву застосовується правило uniqueItems: true, яке в JSON Schema означає перевірку на унікальність (σ_5). Важливо також забезпечити ациклічність ієрархії, тобто, жоден пакет не може включати сам себе, навіть опосередковано. Це перевіряється через обхід графа

включень (наприклад, за допомогою DFS), де контролюється транзитивне замикання відношення contains (σ_3).

Наведемо приклад JSON-фрагменту для пакету UML, який описується як об'єкт, що обов'язково має поля id, name та elements.

```
{
  "$id": "https://example.org/uml/package-schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "UML Package",
  "type": "object",
  "required": ["id", "name", "elements"],
  "properties": {
    "id": {                          // σ1: Унікальні id
      "type": "string",
      "pattern": "^[A-Za-z_][A-Za-z0-9_\\-\\.]*$"
    },
    "name": { "type": "string" }, // Людиночитабельне ім'я (σ2)
    "elements": {
      "type": "array",
      "items": { "oneOf": [
        { "$ref": "class-schema.json" },
        { "$ref": "package-schema.json" }
      ]},
      "uniqueItems": true          // σ5: уникнення структур-дублікатів
    },
    "stereotype": { "type": "string" },
    "tags": {
      "type": "object",
      "additionalProperties": {
        "type": ["string", "number", "boolean"]
      }
    }
  }
}
```

Ідентифікатор задається рядком, що відповідає ^[A-Za-z_][A-Za-z0-9_-.]*$; схема відсікає некоректні значення, а зовнішній валідатор контролює глобальну унікальність. Поле name задає читабельну назву пакета, а окреме правило стежить, аби всередині пакета не було дубльованих назв.

Масив elements через oneOf приймає лише класи та вкладені пакети, тим самим підтримуючи замкненість домену; uniqueItems: true блокує дослівні дублікати, а повторні id перехоплює зовнішній скрипт.

Щоби запобігти рекурсивному включенню пакетів, у CI/CD-конвеєрі запускається обхід графа «пакет містить пакет»; якщо на транзитивному шляху виявляється вузол, що повертається до самого себе, така модель відхиляється.

Поля stereotype та tags є необов'язковими: перше фіксує роль або категорію пакета, друге дозволяє додавати довільні ключ-значення доменної інформації без ризику порушити основні інваріанти.

Уніфіковане подання діаграми компонентів у форматі JSON слугує формальним способом опису архітектурної композиції системи на високому рівні абстракції. Основною метасутністю є Component – логічна одиниця розгортання, яка може мати інтерфейси, залежності, порти, а також пов'язуватись із класами або іншими компонентами.

У JSON-схемі об'єкти компонентів описуються масивом components, де кожен елемент є окремим об'єктом із унікальним id, ім'ям name, та (за потреби) специфікацією interfaces та dependencies. Також передбачена можливість вкладення компонентів, наприклад, для мікросервісної архітектури.

Визначимо загальну структуру JSON-схеми, що описує діаграму компонентів: components – масив компонентів; кожен компонент – об'єкт з полями id, name, interfaces,

dependencies; interfaces – перелік назв або ідентифікаторів точок взаємодії; dependencies – ідентифікатори інших компонентів, від яких залежить даний компонент.

Для забезпечення валідації структура включає обов'язковість ключових полів, верифікацію ідентифікаторів за регулярними виразами, перевірку унікальності та відсутності циклічних залежностей (табл. 4).

Для формалізації моделі компонентів введемо наступні позначення: C – множина всіх компонентів моделі; $id : C \to ID \subset \Sigma^+$ – функція, що відображає компонент у його унікальний ідентифікатор; $name : C \to \Sigma^+$ – функція людиночитабельного імені; $dep \subseteq C \times C$ – бінарне відношення залежностей між компонентами; $iface : C \to 2^{\Sigma^+}$ – функція, яка для кожного компонента визначає множину його інтерфейсів.

Транзитивне замикання dep^+ використовується для визначення наявності непрямих залежностей між компонентами. Воно визначається як об'єднання всіх можливих послідовних застосувань відношення залежності (композиція на себе k разів, $k \geq 1$).

Таблиця 4.

Інваріанти валідації компонентів

№	Назва інваріанта	Формалізація / опис
σ_1	Унікальні ідентифікатори	$\forall c1, c2 \in C : id(c1) = id(c2) \Rightarrow c1 = c2$
σ_2	Унікальні імена	$\forall c1, c2 \in C : name(c1) = name(c2) \Rightarrow c1 = c2$
σ_3	Ациклічність залежностей	$\forall c \in C : c \notin dep^+(c)$
σ_4	Валідність посилань у залежностях	$\forall (c1,c2) \in dep : c1 \in C \wedge c2 \in C$
σ_5	Унікальність інтерфейсів у компоненті	$\forall c \in C$:iface(c) – множина без повторень

Ці обмеження виконуються як частково засобами JSON Schema (σ_1, σ_2, σ_5), так і зовнішніми валідаторами або додатковими пост-обробниками (σ_3, σ_4).

Наведемо приклад опису компонентів у JSON:

```
{
    "components": [{
            "id": "AuthService",
            "name": "Authentication",
            "interfaces": ["login", "logout", "tokenValidate"],
            "dependencies": ["UserDB"]
        }, {
            "id": "UserDB",
            "name": "User Database",
            "interfaces": ["readUser", "writeUser"]
        }
    ]
}
```

У цьому прикладі описано два компоненти: AuthService і UserDB. Кожен має унікальний id та читабельну назву name. Компонент AuthService реалізує інтерфейси login, logout, tokenValidate і залежить від UserDB. Всі поля відповідають правилам: id є строковим значенням, що задовольняє регулярний вираз, масиви інтерфейсів не містять повторень, і залежність не породжує циклів. Структура відповідає інваріантам σ_1–σ_5 і готова до автоматизованої перевірки.

У випадку діаграми об'єктів основна мета JSON-схеми полягає в представленні конкретних екземплярів класів, що відповідають об'єктам моделі. Така схема є доповненням до діаграми класів і забезпечує семантично точне відображення стану системи на певному етапі її виконання або проектування.

JSON-схема для діаграми об'єктів містить масив objects, кожен елемент якого описує окремий екземпляр класу. Об'єкт обов'язково включає поля: id – унікальний ідентифікатор екземпляра; class – посилання на відповідний клас (за id класу з UML-моделі); slotValues – об'єкт, що є словником значень атрибутів, де ключ – це ім'я

атрибута, а значення – значення цього атрибута; links – необов'язковий масив, що задає зв'язки з іншими об'єктами (відповідає асоціаціям).

Крім того, для збереження узгодженості моделі застосовуються обмеження: кожен class має бути валідним посиланням на визначений у діаграмі класів тип; усі slotValues повинні відповідати типам атрибутів, що описані в класі; об'єкти, що беруть участь у links, мають бути оголошені у межах цієї ж діаграми; значення id кожного об'єкта має бути унікальним у межах всієї діаграми.

Для формалізації моделі об'єктів введемо наступні позначення: O – множина об'єктів; C – множина класів із UML-діаграми; A(c) – множина атрибутів класу c∈C; V – множина значень; $id : O \to ID \subseteq \Sigma^+$ – функція, що задає унікальний ідентифікатор об'єкта; $class : O \to C$; $value : O \times A(class(o)) \to V$ – функція, що визначає значення кожного атрибута; $link \subseteq O \times O$ – бінарне відношення між об'єктами (відповідає асоціаціям).

Система обмежень формалізована у таблиці інваріантів, яка забезпечує унікальність ідентифікаторів, відповідність атрибутів класам, коректність типів значень та відсутність посилань на неіснуючі об'єкти (табл. 5).

Таблиця 5.

Інваріанти валідації об'єктів

№	Назва інваріанта	Формалізація / опис
1	Унікальність id	$\forall o_1, o_2 \in O: id(o_1) = id(o_2) \Rightarrow o_1 = o_2$
2	Належність класу	$\forall o \in O: class(o) \in C$
3	Наявність значень для атрибутів	$\forall o \in O, \forall a \in A(class(o)): \exists v \in V: value(o, a) = v$
4	Коректність посилань	$\forall (o_1, o_2) \in link: o_1 \in O \land o_2 \in O$
5	Відповідність типів значень	$\forall o \in O, \forall a \in A(class(o)): type(value(o, a)) \in allowed\_types(a)$

Наведемо JSON-приклад об'єктів obj1 і obj2, що є екземплярами класу Person, який повинен бути визначений у діаграмі класів з атрибутами name та age.

```
{
 "objects":[
 {
       "id":"obj1",
       "class":"Person",
       "slotValues":{"name":"Alice","age":30},
       "links":["obj2"]
 },
 {
       "id":"obj2",
       "class":"Person",
       "slotValues":{"name":"Bob","age":35},
       "links":["obj1"]
 }]
}
```

Значення атрибутів відповідають очікуваним типам (string, integer). Поле links відображає взаємозв'язок між об'єктами, що в UML відповідає бінарній асоціації між екземплярами класу.

Такий підхід забезпечує формальну відповідність між класовою і об'єктною діаграмами, дозволяючи проводити типову й структурну валідацію даних, підтримувати тестування моделей та їх верифікацію в інструментах UML-моделювання.

Для уніфікованої реалізації перевірки відповідності UML-моделей у JSON-форматі нижче подано узагальнену таблицю валідаційних правил, що реалізуються на рівні JSON Schema або зовнішнього валідатора (табл. 6).

Таблиця 6.

Узагальнення валідаційних правил

№	Група правил	Формальна/схемна реалізація	Призначення
R-1	Обов'язкові поля	required у кореневих та вкладених об'єктах minItems: 1 для масивів classes, packages, components, objects	Гарантія мінімально необхідної інформації; порожній масив класів або відсутність id у елемента – критична помилка
R-2	Універсальний ідентифікатор	json pattern: "^[A-Za-z_][A-Za-z0-9_\\-\\.]*$" propertyNames для словників (напр. тегів)	Єдина регулярна форма id для всіх типів елементів. Літери, цифри, підкреслення, дефіс, крапка; починатися має з літери/_
R-3	Перелік допустимих примітивних типів	json { "enum": ["Boolean","Integer","Real","String","Date","Time"] }	Вирівнює типи атрибутів/параметрів з UML 2.5; запобігає довільним або помилковим позначенням (int, str, …)
R-4	Унікальність у масивах	uniqueItems: true та/або користувацький ключ uniqueItemProperties: ["id"] (підтримується бібліотекою Ajv)	Забороняє дублювати елементи в межах колекції (класи в пакеті, атрибути в класі, інтерфейси в компоненті)
R-5	Крос-посилання ($ref/$data)	перевірка типу: "$ref": "#/$defs/ClassId" короткий $data-rule: "not": { "const": { "$data": "1/clientId" } }	Виконує семантичну узгодженість: кожне class, supplierId, superClassId тощо має співпадати з реально оголошеним id
R-6	Валідація імен (людиночитабельних)	json pattern: "^[A-Z][A-Za-z0-9_]*$" (допустимі також локалізовані літери)	Уникає порожніх/невалідних назв; полегшує читабельність у графічних редакторах
R-7	Ациклічність ієрархій	зовнішній алгоритм обходу графа DFS / топологічне сортування для contains, generalization, dependency	Логічна перевірка, яку складно записати чистими ключами JSON Schema; виконується у скрипті CI/CD
R-8	Контроль типів значень у діаграмі об'єктів	oneOf / if…then…else на основі $data : якщо атрибут оголошено як "Integer" → type: "integer", якщо "String" → type: "string"	Забезпечує, щоб екземпляр об'єкта реально дотримувався типу, визначеного в класі
R-9	Розширюваність (стереотипи, теги)	поле tags описується як "additionalProperties": { "type": ["string","number","boolean"] }	Дозволяє довільно додавати метадані без зміни базової схеми; зберігає forward-compatibility

Результати та обговорення. Аналіз перевірки UML-моделей показав, що близько 70 % інваріантів Σ реалізуються засобами JSON Schema (required, enum, pattern, uniqueItems, if…then…else), що охоплює перевірку синтаксису, структури й типізації. Проте для 30 % складніших залежностей (ациклічність відношень успадкування та включення, транзитивну узгодженість зв'язків, цілісність міжоб'єктних посилань, унікальність імен у вкладених просторах) стандартної декларативної виразності недостатньо.

Для таких перевірок застосовується розширена валідація через зовнішні процедури (after-hooks) або CI/CD-механізми, які дозволяють інтерпретувати модель як граф і виконати обхід (наприклад, DFS чи топологічне сортування) для підтвердження інваріантів, що потребують глобального або транзитивного контексту. У межах цих процедур: JSON-файл з моделлю парситься у вигляді графу; окремі модулі перевіряють інваріанти σ_i, які потребують глобального контексту; результати перевірок повертаються у вигляді узагальненого звіту або блокують конвеєр, якщо виявлено структурні помилки.

Висновки. У роботі сформульовано та реалізовано формалізоване структурне подання UML-метамоделі у форматі JSON, що охоплює основні метасутності (класи, атрибути, асоціації, узагальнення, пакети, компоненти, об'єкти) та відповідні відношення між ними. Запропоноване подання базується на системі JSON Schema з урахуванням формальних інваріантів структурної цілісності, які задають правила іменування, вкладеності, типізації та відсутності циклів.

Практична апробація моделі показала, що більшість інваріантів може бути реалізована стандартними засобами JSON Schema, однак низка критичних перевірок (зокрема, транзитивні й міжсхемні залежності) потребує зовнішніх процедур контролю. Це зумовлює необхідність інтеграції розширених валідаційних механізмів у середовища CI/CD або спеціалізовані інструменти перевірки UML-моделей.

Запропоноване подання метамоделі створює підґрунтя для подальших досліджень у напрямках: автоматизованої верифікації цілісності структурних моделей; трансформації UML-діаграм у мови програмування або формати XMI; побудови відкритого інструментарію для аналізу й редагування UML-моделей у JSON форматі.

Список літератури

1. Komleva N.O., Nikitchenko M.I. Method for incremental control of consistency between structural and behavioral views of software architecture. Applied Aspects of Information Technology. 2025. Vol. 8, No. 2. (в друці) DOI: https://doi.org/10.15276/aait.08.2025.6.
2. Nikitchenko M.I. Analysis of formats for storing UML models in modern CASE tools. Technology and Society: Interaction, Influence, Transformation: Proceedings of the IV International Scientific Conference (20 June 2025, Chernihiv, Ukraine). 2025. P. 208–212. DOI: https://doi.org/10.62731/mcnd-20.06.2025.
3. Clarisó R., González C.A., Cabot J. Incremental verification of UML/OCL models using logical abstraction. Journal of Object Technology. 2020. Vol. 19, No. 3. P. 1–20. DOI: https://doi.org/10.5381/jot.2020.19.3.a7.
4. Shaikh A., Wiil U. K. Overview of slicing and feedback techniques for efficient verification of UML/OCL class diagrams. IEEE Access. 2018. Vol. 6. P. 23864–23882. DOI: https://doi.org/10.1109/ACCESS.2018.2797695.
5. Wright A., Andrews H., Hutton B. JSON Schema: A media type for describing JSON documents [Електронний ресурс]. 2019. URL: https://json-schema.org/draft/2019-09/json-schema-core.html.
6. Wright A., Andrews H., Hutton B. JSON Schema Validation: A vocabulary for structural validation of JSON [Електронний ресурс]. 2020. URL: https://json-schema.org/draft/2020-12/json-schema-validation.html.
7. Portele C., Echterhoff J., et al. Best practice for OGC – UML to JSON encoding rules [Електронний ресурс]. 2020. URL: http://www.opengis.net/doc/BP/uml-to-json-encoding/1.0.
8. Attouche L., Baazizi M.A., Colazzo D., Ghelli G. Validation of modern JSON Schema: formalization and complexity. arXiv preprint. 2023. DOI: https://doi.org/10.48550/arXiv.2307.10034.
9. Attouche L., Baazizi M.A., Colazzo D., Ulliana F. Elimination of annotation dependencies in validation for modern JSON Schema. arXiv preprint. 2025. arXiv: 2503.11288. URL: https://arxiv.org/abs/2503.11288.

10. Khan A.H., Porres I. Consistency of UML class, object and statechart diagrams using ontology reasoners. arXiv preprint. 2022. arXiv: 2205.11177. URL: https://arxiv.org/abs/2205.11177.
11. Shaikh A., Khan A. H., Wiil U. S. More than two decades of research on verification of UML class models: A systematic literature review. IEEE Access. 2021. Vol. 9. P. 128266–128295. DOI: https://doi.org/10.1109/ACCESS.2021.3121222.
12. SOM-Research. JSONSchema to UML [Електронний ресурс]. 2022. URL: https://github.com/SOM-Research/jsonschema-to-uml.
13. StarUML Documentation. StarUML model file structure [Електронний ресурс]. 2024. URL: https://docs.staruml.io/user-guide/managing-project.
14. Modelio Documentation. Modelio JSON export format [Електронний ресурс]. 2023. URL: https://www.modelio.org/index.htm.
15. Hafeez A., Musavi H. A., Rehman A.-u. Ontology-Based Transformation and Verification of UML Class Model. The International Arab Journal of Information Technology. 2020. Vol. 17, No. 5. P. 758–768. DOI: https://doi.org/ 10.34028/iajit/17/5/9.
16. Di Ruscio D., Franzago M., Malavolta I., Muccini H. Envisioning the Future of Collaborative Model-Driven Software Engineering. 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina. 2017. P. 173–175. DOI: https://doi.org/10.1109/ICSE-C.2017.143.
17. Döller V. Formalizing the four-layer metamodeling stack with MetaMorph: potential and benefits. Software and Systems Modeling. 2022. Vol. 21. P. 1411–1435. DOI: https://doi.org/10.1007/s10270-022-00986-2.

STRUCTURAL VIEW OF THE UML METAMODEL IN JSON FORMAT

N.O. Komleva, M.I. Nikitchenko

Odesa Polytechnic National University
1, Shevchenko Ave., Odesa, 65044, Ukraine
Emails: komleva@op.edu.ua, maksym.nikitchenko@gmail.com

The paper considers the problem of formal representation of UML (Unified Modeling Language) models in an open format suitable for verification, transformation, and integration into software development engineering processes. The conceptual basis of the study is the UML metamodel, which includes two interrelated views: structural (defines objects and model relationships) and behavioral (describes activities, states, and sequences). This metamodel serves as the basis for incremental control of software architecture consistency. In this work, special attention is paid to the structural view of the UML metamodel, which is the basis for representing the system architecture in the form of interrelated object types. For this view, a formalized representation of key UML meta-entities, such as classes, attributes, associations, packages, components, and objects, is implemented in JSON (JavaScript Object Notation) format. The chosen view is based on the application of the JSON Schema standard, which allows for a clear description of valid structures, data types, named properties, and validation rules. As a result, a complete set of types and relationships has been constructed, which defines the allowed configuration of the UML model as a structured set of objects with clearly defined relationships between them. To support the syntactic and semantic integrity of this model, a system of formalized invariants has been formulated, which fixes critical constraints, in particular, requirements for identifiers, correct nesting of elements, strict typing of attributes, and the absence of cyclic dependencies in composite and inherited structures. This lays the foundation for a unified representation of structural models and further automated verification of their correctness. Some of the rules are implemented by the JSON schema itself, while more complex ones are implemented through external verification mechanisms. The proposed approach ensures a clear correspondence between abstract UML objects and their representation in an open format, which, in the long term, opens up the possibility of unified integration with other modeling tools, CI/CD, and code generation tools. The research focuses on formalism, transparency, and consistency with the UML specification, which are necessary to support incremental consistency control while ensuring compatibility between the components of the metamodel.
Keywords: modeling, UML, metamodel, JSON Schema, validation, invariants.