НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ПОЛІТЕХНІКА»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ПОЛІТЕХНІКА»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Кваліфікаційна наукова
праця на правах рукопису

НІКІТЧЕНКО МАКСИМ ІГОРОВИЧ

УДК 004.942: 004.052

ДИСЕРТАЦІЯ

МЕТОДИ ЗАБЕЗПЕЧЕННЯ КОНСИСТЕНТНОСТІ МОДЕЛЕЙ ПРОГРАМНИХ СИСТЕМ В ІНКРЕМЕНТАЛЬНИХ ПРОЦЕСАХ РОЗРОБКИ ТА ЗАСОБИ ЇХ ІНТЕГРАЦІЇ З ІНСТРУМЕНТАЛЬНИМИ СЕРЕДОВИЩАМИ

Спеціальність: 121 Інженерія програмного забезпечення
Галузь знань: 12 Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.
_____ М. І. Нікітченко

Науковий керівник – Комлева Н. О. кандидат технічних наук, доцент

Одеса – 2025

АНОТАЦІЯ

Нікітченко М. І. Методи забезпечення консистентності моделей програмних систем в інкрементальних процесах розробки та засоби їх інтеграції з інструментальними середовищами. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 121 Інженерія програмного забезпечення. – Національний університет «Одеська політехніка» Міністерства освіти і науки України, Одеса, 2025.

Роботу присвячено вирішенню актуальної науково-прикладної задачі, яка полягає у підвищенні ефективності та достовірності процесу модельно-орієнтованої розробки складних програмних систем шляхом розробки науково-методичного апарату та інструментальних засобів для забезпечення гарантованої консистентності UML-моделей.

Актуальність теми дослідження визначається наявним протиріччям між вимогами до формальної строгості моделей UML, яка традиційно забезпечується стандартом XMI, та потребою у гнучкості й інтеграції з сучасними ітеративними процесами розробки, де перевагу мають легковагові формати, такі як JSON. Цей конфлікт призводить до розриву між проектною моделлю та її реалізацією, провокуючи архітектурний дрейф та накопичення прихованих архітектурних дефектів, що експоненційно збільшує вартість супроводу програмного забезпечення. Встановлене протиріччя долається шляхом розробки нового підходу, що поєднує гнучкість формату JSON для опису статичної структури та формальну строгість XMI для поведінкової семантики в рамках єдиної, цілісної та гарантовано узгодженої моделі з двома поданнями. Цей підхід забезпечує ефективну перевірку консистентності в умовах сучасної ітеративної розробки. Це набуває особливого значення в контексті автоматизації процесів DevOps, дозволяючи подолати ізольованість класичних CASE-засобів від конвеєрів CI/CD. Відтак, запропоноване рішення уможливлює реалізацію стратегії «shift-left», виявляючи архітектурні помилки на ранніх стадіях без сповільнення темпів написання коду.

Метою роботи є підвищення ефективності та достовірності процесу модельно-орієнтованої розробки складних програмних систем шляхом розробки науково-методичного апарату та інструментальних засобів для забезпечення гарантованої консистентності UML-моделей з двома поданнями (структурним у форматі JSON та поведінковим у форматі XMI).

Для досягнення поставленої мети було сформульовано та вирішено такі **задачі**:

– Проведено аналіз проблеми забезпечення консистентності UML-моделей, існуючих форматів їх зберігання та функціональності сучасних CASE-засобів для виявлення їх існуючих недоліків та обґрунтування комбінованого підходу зі структурним та поведінковим поданнями. Додатково було сформульовано формальну класифікацію помилок неузгодженості між структурними та поведінковими поданнями для обґрунтування актуальності проблеми.

– Розроблено формальну метамодель UML з двома поданнями, що математично описує структурне та поведінкове подання, та встановлено відношення відповідності між ними. Для структурного подання було сформовано узагальнений підхід до його відображення, ідентифікації елементів та дворівневої валідації у JSON; для поведінкового подання так само були визначені та описані узагальнені правила його представлення через профіль UML, трасування до структури та перевірки формальних інваріантів у XMI. Додатково, для цієї метамоделі було сформульовано набір правил консистентності мовою OCL та обґрунтовано їх трансформацію до формалізму, придатного для автоматизованої верифікації за допомогою Alloy.

– Розроблено метод автоматизованого забезпечення консистентності, який базується на комбінованому застосуванні OCL-інваріантів для перевірки правил та трансформації до формалізму Alloy для глибокої верифікації. Сформульовано та доведено теорему інкрементальної консистентності, яка теоретично обґрунтовує метод інкрементальної валідації на основі графа залежностей з класифікованими ребрами. Також розроблено метод двосторонньої синхронізації та алгоритми трансформації між монолітним поданням у форматі XMI та запропонованою моделлю з двома поданнями для забезпечення сумісності з існуючими інструментами.

– Розроблено архітектуру та реалізовано прототип програмного плагіна, що

втілює запропоновані методи, та проведено експериментальну оцінку його ефективності й продуктивності у порівнянні з існуючими промисловими аналогами.

Наукова новизна одержаних результатів, які виносяться на захист, полягає у наступному.

– *Вперше розроблено* формальну метамодель UML, яка охоплює структурне та поведінкове подання, реалізовані на основі комбінованого використання форматів JSON та XMI відповідно, які логічно пов’язані між собою через формально визначене відношення відповідності; для цих подань сформульовано та узагальнено підходи до їх формалізації (для JSON) та правил представлення й валідації (для XMI). На відміну від існуючих підходів, що використовують монолітні або пропрієтарні формати, запропонована метамодель одночасно забезпечує компактне та VCS-дружнє зберігання статичної структури, адаптовану до частих і швидких змін, та підтримку стандартного XMI-представлення для динамічних аспектів, що забезпечує як сумісність із існуючими інструментами моделювання, так і ефективну інтеграцію в сучасні ітеративні процеси розробки.

– *Вперше запропоновано* метод автоматизованого забезпечення консистентності UML-моделі, побудованої за принципом двох подань. Метод функціонує на основі формально визначеного відношення відповідності між структурними та поведінковими елементами та базується на застосуванні OCL-інваріантів для верифікації локальної та глобальної узгодженості моделей при інкрементальному редагуванні. Для подолання обмежень OCL, він також включає двоступеневий підхід з трансформацією моделі у специфікацію Alloy для глибокої верифікації за допомогою SAT-вирішувачів. На відміну від відомих методів валідації, запропонований метод дозволяє проводити глибоку формальну верифікацію складних транзитивних властивостей та автоматично генерувати контрприклади для дефектів, що підвищує точність і повноту виявлення семантичних неузгодженостей.

– *Вперше сформульовано та доведено* теорему інкрементальної консистентності, яка гарантує, що будь-яка послідовність допустимих змін над структурним або поведінковим поданням, за умови виконання локалізованих перевірок OCL-інваріантів, зберігає глобальну узгодженість моделі в межах

формального відношення відповідності. На відміну від традиційних підходів повної перевірки (зі складністю O(N)), це дозволяє теоретично обґрунтувати коректність та обчислювальну ефективність (зі складністю O(k·d)) інкрементальних методів валідації, завдяки чому можливе створення інструментів, придатних для інтеграції в CI/CD та IDE.

– *Отримав подальший розвиток* метод двосторонньої синхронізації, що реалізований через механізм інкрементальних оновлень та локалізованої перевірки консистентності, яка ґрунтується на формальному графі залежностей із запропонованою класифікацією його ребер на структурні, семантичні та трасувальні. Особливістю розробленого методу є поєднання автоматичного виявлення конфліктів, формалізованих правил трансформації моделей і підтримки взаємної актуальності обох подань у реальному часі. На відміну від існуючих підходів, що часто є односпрямованими або покладаються на ручне втручання, запропонований метод забезпечує проактивну підтримку узгодженості в обох напрямках, що дозволяє одночасно вносити зміни як з боку коду, так і з боку моделі, запобігаючи архітектурному дрейфу.

Практичне значення одержаних результатів полягає у розробці алгоритмічних та інструментальних засобів, що може бути безпосередньо впроваджено в процеси розробки програмного забезпечення. Розроблено та реалізовано прототип програмного плагіна, який втілює запропоновані методи, може бути інтегрований у сучасні процеси безперервної інтеграції та доставки і дозволяє автоматизувати контроль архітектурної цілісності. Запропонований підхід дозволяє скоротити час перевірки консистентності після внесення змін у 12 разів (з 3,8 с до 0,3 с) та зменшити споживання оперативної пам’яті вдвічі порівняно з традиційними методами повної валідації. Експериментально підтверджено високу достовірність методу: точність виявлення семантичних дефектів (F1-міра) становить 98,5 % у нативному середовищі та 95,7 % в умовах інтероперабельності, що значно перевищує показники існуючих промислових аналогів. В результаті це дозволяє знизити вартість та трудомісткість розробки за рахунок раннього автоматизованого виявлення прихованих архітектурних дефектів, запобігаючи їх прояву на пізніх стадіях життєвого циклу.

Об'єкт дослідження – процес модельно-орієнтованої розробки програмного забезпечення.

Предмет дослідження – моделі та інструментальні засоби забезпечення, які реалізують методи консистентності UML-моделей зі структурним та поведінковим поданнями.

Ключові слова: модельно-орієнтована інженерія, UML, консистентність моделі, подання моделі, JSON, XMI, інкрементальна валідація, двостороння синхронізація, програмний плагін, граф залежностей.

Список публікацій здобувача

Наукові праці, в яких опубліковано основні наукові результати дисертації.

1. Нікітченко М. І. Дворівнева архітектура UML на основі гібридного формату JSON IXMI. Вчені записки ТНУ імені В.І. Вернадського. Серія: Технічні науки. 2025. Т. 36 (75), № 1. С. 157–162. DOI: 10.32782/2663-5941/2025.1.2/23. *Видання включено до переліку наукових фахових видань України, категорія «Б».*

https://tech.vernadskyjournals.in.ua/journals/2025/1_2025/part_2/25.pdf

2. Nikitchenko, M. I., Komleva, N. O. Method for Incremental Control of Consistency Between Structural and Behavioral Views of Software Architecture. AAIT. 2025, 8 (2), 162–177. DOI: 10.15276/aait.08.2025.11. *Видання включено до переліку наукових фахових видань України, категорія «Б».*

https://aait.od.ua/index.php/journal/article/view/177/179

3. Нікітченко М. І., Комлева Н. О. Структурне подання UML-метамоделі у форматі JSON. Інформатика та математичні методи в моделюванні. 2025. Том 15, № 2. С. 205–217. DOI: 10.15276/imms.v15.no2.205. *Видання включено до переліку наукових фахових видань України, категорія «Б».*

http://immm.op.edu.ua/files/archive/n2_v15_2025/2025_2(6).pdf

4. Нікітченко М. І., Комлева Н. О. Поведінкове подання у форматі XMI в межах UML-метамоделі з двома поданнями. Herald of Khmelnytskyi National University. Technical sciences. 2025. Т. 357, № 5.1, С. 326-336. DOI: 10.31891/2307-5732-2025-357-42. *Видання включено до переліку наукових фахових видань України, категорія «Б».*

https://heraldts.khmnu.edu.ua/index.php/heraldts/article/view/1943

5. Нікітченко М. І. Архітектура та реалізація плагіна інкрементальної валідації UML-метамоделей з двома поданнями. Вчені записки ТНУ імені В.І. Вернадського. Серія: Технічні науки. 2025. Т. 36 (75), № 4. С. 208-216. DOI: 10.32782/2663-5941/2025.4.2/28. *Видання включено до переліку наукових фахових видань України, категорія «Б».*

https://tech.vernadskyjournals.in.ua/journals/2025/4_2025/part_2/30.pdf

6. Нікітченко М. І., Комлева Н. О. Метод двосторонньої синхронізації UML-моделі з двома поданнями на основі інкрементальних оновлень. Вісник Херсонського національного технічного університету. 2025. Т. 3(94), № 2. С. 243-249. DOI: 10.35546/kntu2078-4481.2025.3.2.30. *Видання включено до переліку наукових фахових видань України, категорія «Б».*

https://journals.kntu.kherson.ua/index.php/visnyk_kntu/article/view/1137

Наукові праці, які засвідчують апробацію матеріалів дисертації.

7. Нікітченко М. І. Управління та редагування UML-документів: структура, інтеграція, автоматизація. Інформатика. Культура. Техніка. Одеса, 2024. Т. 1. № 1. С. 104-111. DOI: 10.15276/ict.01.2024.15.

https://ict.op.edu.ua/index.php/journal/uk/article/view/105

8. Нікітченко М. І. Розвиток і вдосконалення UML-моделей у гібридному форматі. Матеріали конференцій МЦНД. Дрогобич, Україна, 31.01.2025. С. 294–296. DOI: 10.62731/mcnd-31.01.2025.008.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/521

9. Нікітченко М. І. Інтеграція гібридного формату UML та IDE в контексті Model-Driven Development. Матеріали конференцій МЦНД. Тернопіль, Україна, 21.03.2025. С. 192–194. DOI: 10.62731/mcnd-21.03.2025.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/654

10. Нікітченко М. І. Аналіз форматів збереження UML-моделей у сучасних CASE-засобах. Матеріали конференцій МЦНД. Чернігів, Україна, 20.06.2025. С. 208–212. DOI: 10.62731/mcnd-20.06.2025.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/862

11. Нікітченко М. І. Механізм перетворення UML-моделей із XMI до подання з розділенням структури та поведінки. Матеріали конференцій МЦНД. Рівне, Україна, 12.09.2025. С. 108-111. DOI: 10.62731/mcnd-12.09.2025.004.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/1047

12. Нікітченко М. І. Механізм зворотної трансформації комбінованої UML-моделі у формат стандартного XMI. Матеріали конференцій МЦНД. Тернопіль, Україна, 19.09.2025. С. 94-97. DOI: 10.62731/mcnd-19.09.2025.004

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/1053

ABSTRACT

Nikitchenko M. I. Methods for ensuring the consistency of software system models in incremental development processes and means of their integration with tool environments. – Qualification scientific work on the rights of the manuscript.

Thesis for the degree of Doctor of Philosophy in the specialty 121 Software Engineering. – National University “Odessa Polytechnic” of the Ministry of Education and Science of Ukraine, Odessa, 2025.

The work is devoted to solving an urgent scientific and applied problem, which consists in increasing the efficiency and reliability of the model-oriented development of complex software systems by developing a scientific and methodological apparatus and tools to ensure the guaranteed consistency of UML models.

The relevance of the research topic is determined by the existing contradiction between the requirements for the formal rigor of UML models, which is traditionally ensured by the XMI standard, and the need for flexibility and integration with modern iterative development processes, where lightweight formats such as JSON are preferred. This conflict leads to a gap between the design model and its implementation, provoking “architectural drift” and the accumulation of hidden architectural defects, which exponentially increases the cost of software maintenance. The established contradiction is overcome by developing a new approach that combines the flexibility of the JSON format for describing static structure and the formal rigor of XMI for behavioral semantics within

a single, coherent, and guaranteed consistent model with two views. This approach provides effective consistency checking in the context of modern iterative development. This is particularly important in the context of DevOps process automation, as it allows overcoming the isolation of classic CASE tools from CI/CD pipelines. Thus, the proposed solution enables the implementation of the «shift-left» strategy, detecting architectural errors at early stages without slowing down the pace of code writing.

The purpose of the study is to improve the efficiency and reliability of the model-driven development process for complex software systems by developing scientific and methodological tools and instruments to ensure the guaranteed consistency of UML models with two viws (structural in JSON format and behavioral in XMI format).

To achieve this goal, the following tasks were formulated and solved:

– An analysis of the problem of ensuring the consistency of UML models, existing formats for their storage, and the functionality of modern CASE tools was conducted to identify their existing shortcomings and justify a combined approach with structural and behavioral representations. In addition, a formal classification of inconsistencies between structural and behavioral viws was formulated to justify the relevance of the problem.

– A formal UML metamodel with two viws was developed, which mathematically describes the structural and behavioral views, and the correspondence between them was established. For the structural view, a generalized approach to its reflection, element identification, and two-level validation in JSON was formulated; for the behavioral view, generalized rules for its representation through the UML profile, tracing to the structure, and verification of formal invariants in XMI were also defined and described. Additionally, a set of consistency rules was formulated for this metamodel in OCL and their transformation into a formalism suitable for automated verification using Alloy was justified.

– A method for automated consistency assurance has been developed, based on the combined use of OCL invariants for rule checking and transformation to Alloy formalism for deep verification. An incremental consistency theorem has been formulated and proven, which theoretically justifies the method of incremental validation based on a dependency graph with classified edges. A method of bidirectional synchronization and transformation algorithms between a monolithic representation in XMI format and the proposed model with

two representations have also been developed to ensure compatibility with existing tools.

– An architecture has been developed and a prototype software plugin implementing the proposed methods has been implemented, and an experimental evaluation of its effectiveness and performance has been conducted in comparison with existing industrial analogues.

The scientific novelty of the results presented for defense lies in the following:

– *For the first time*, a formal UML metamodel has been developed that covers structural and behavioral views implemented based on the combined use of JSON and XMI formats, respectively, which are logically linked through a formally defined correspondence relationship. Approaches to their formalization (for JSON) and representation and validation rules (for XMI) have been formulated and generalized for these views. Unlike existing approaches that use monolithic or proprietary formats, the proposed metamodel simultaneously provides compact and VCS-friendly storage of static structure, adapted to frequent and rapid changes, and support for standard XMI representation for dynamic aspects, ensuring both compatibility with existing modeling tools and effective integration into modern iterative development processes.

– *For the first time*, a method for automated consistency assurance of a UML model built on the principle of two views is proposed. The method operates on the basis of a formally defined correspondence between structural and behavioral elements and is based on the use of OCL invariants to verify the local and global consistency of models during incremental editing. To overcome the limitations of OCL, it also includes a two-step approach with model transformation into an Alloy specification for deep verification using SAT solvers. Unlike known validation methods, the proposed method allows for deep formal verification of complex transitive properties and automatic generation of counterexamples for defects, which increases the accuracy and completeness of semantic inconsistency detection.

– *For the first time*, the incremental consistency theorem has been formulated and proven, which guarantees that any sequence of permissible changes to the structural or behavioral view, subject to localized checks of OCL invariants, preserves the global consistency of the model within the formal correspondence relation. Unlike traditional full

verification approaches (with O(N) complexity), this allows us to theoretically justify the correctness and computational efficiency (with O(k·d) complexity) of incremental validation methods, making it possible to create tools suitable for integration into CI/CD and IDE.

– The method of two-way synchronization *has been further developed*, implemented through a mechanism of incremental updates and localized consistency checking, which is based on a formal dependency graph with the proposed classification of its edges into structural, semantic, and tracing ones. A distinctive feature of the developed method is the combination of automatic conflict detection, formalized model transformation rules, and real-time support for the mutual relevance of both representations. Unlike existing approaches, which are often unidirectional or rely on manual intervention, the proposed method provides proactive support for consistency in both directions, allowing simultaneous changes to be made on both the code and model sides, preventing architectural drift.

The practical significance of the obtained results lies in the development of algorithmic and instrumental means that can be directly implemented in software development processes. A prototype software plugin has been developed and implemented that embodies the proposed methods, can be integrated into modern continuous integration and delivery processes, and allows for the automation of architectural integrity control. The proposed approach reduces the consistency verification time after changes by 12 times (from 3.8 s to 0.3 s) and halves the RAM consumption compared to traditional full validation methods. The high reliability of the method has been experimentally confirmed: the accuracy of semantic defect detection (F1-measure) is 98.5% in a native environment and 95.7% in interoperability conditions, which significantly exceeds the performance of existing industrial analogues. As a result, this reduces the cost and labor intensity of development through early automated detection of hidden architectural defects, preventing their manifestation in later stages of the life cycle.

The object of research is the process of model-driven software development.

The subject of research is methods, models, and tools for ensuring the consistency of UML models with two representations.

Keywords: model-driven engineering, UML, model consistency, two model views,

JSON, XMI, incremental validation, two-way synchronization, software plugin, dependency graph.

List of publications of the applicant

Scientific works in which the main scientific results of the dissertation are published.

1. Nikitchenko, M. I. Two-tier UML architecture based on hybrid JSON and XMI format. Scientific Notes of V. I. Vernadsky Ternopil National University. Series: Technical Sciences. 2025. Vol. 36 (75), No. 1. Pp. 157–162. DOI: 10.32782/2663-5941/2025.1.2/23. *The issue is included in the list of scientific professional publications of Ukraine, category «B».*

https://tech.vernadskyjournals.in.ua/journals/2025/1_2025/part_2/25.pdf

2. Nikitchenko, M. I., Komleva, N. O. Method for Incremental Control of Consistency Between Structural and Behavioral Views of Software Architecture. AAIT. 2025, 8 (2), 162–177. DOI: 10.15276/aait.08.2025.11. *The issue is included in the list of scientific professional publications of Ukraine, category «B».*

https://aait.od.ua/index.php/journal/article/view/177/179

3. Nikitchenko M. I., Komleva N. O. Structural view of the UML metamodel in JSON format. Informatics and Mathematical Methods in Modeling. 2025. Volume 15, No. 2. Pp. 205–217. DOI: 10.15276/imms.v15.no2.205. *The issue is included in the list of scientific professional publications of Ukraine, category «B».*

http://immm.op.edu.ua/files/archive/n2_v15_2025/2025_2(6).pdf

4. Nikitchenko M. I., Komleva N. O. Behavioral view in XMI format within a UML metamodel with two views. Herald of Khmelnytskyi National University. Technical sciences. 2025. Vol. 357, No 5.1, Pp. 326-336. DOI: 10.31891/2307-5732-2025-357-42. *The issue is included in the list of scientific professional publications of Ukraine, category «B».*

https://heraldts.khmnu.edu.ua/index.php/heraldts/article/view/1943

5. Nikitchenko M. I. Architecture and implementation of a plugin for incremental validation of UML metamodels with two views. Scientific notes of V.I. Vernadsky Ternopil National University. Series: Technical sciences. 2025. Vol. 36 (75), No. 4. Pp. 208-216.

DOI: 10.32782/2663-5941/2025.4.2/28. *The issue is included in the list of scientific professional publications of Ukraine, category «B».*

https://tech.vernadskyjournals.in.ua/journals/2025/4_2025/part_2/30.pdf

6. Nikitchenko M. I., Komleva N. O. Method for bilateral synchronization of a UML model with two representations based on incremental updates. Bulletin of Kherson National Technical University. 2025. Vol. 3(94), No. 2. Pp. 243-249. DOI: 10.35546/kntu2078-4481.2025.3.2.30. *The issue is included in the list of scientific professional publications of Ukraine, category «B».*

https://journals.kntu.kherson.ua/index.php/visnyk_kntu/article/view/1137

Scientific works that confirm the approbation of the dissertation materials.

7. Nikitchenko M. Management and editing of UML documents: structure, integration, automation. Informatics. Culture. Technics. Odesa, 2024. Vol. 1, no. 1. P. 104–111. DOI: 10.15276/ict.01.2024.15.

https://ict.op.edu.ua/index.php/journal/uk/article/view/105

8. Nikitchenko M. Development and improvement of UML models in a hybrid format. Conference Proceedings of the International Center for Scientific Research. Drohobych, Ukraine, January 31, 2025. P. 294–296. DOI: https://doi.org/10.62731/mcnd-31.01.2025.008.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/521

9. Nikitchenko M. Integration of the hybrid UML format and IDE in the context of Model-Driven Development. Conference Proceedings of the International Center for Scientific Research. Ternopil, Ukraine, March 21, 2025. P. 192–194. DOI: 10.62731/mcnd-21.03.2025.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/654

10. Nikitchenko M. Analysis of storage formats for UML models in modern CASE-tools. Conference Proceedings of the International Center for Scientific Research. Chernihiv, Ukraine, June 20, 2025. P. 208–212. DOI: 10.62731/mcnd-20.06.2025.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/862

11. Nikitchenko M. Mechanism of converting UML models from XMI to a

representation with separated structure and behavior. Conference Proceedings of the International Center for Scientific Research. Rivne, Ukraine, September 12, 2025. P. 108–111. DOI: 10.62731/mcnd-12.09.2025.004.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/1047

12. Nikitchenko M. Mechanism of reverse transformation of a hybrid UML model into the standard XMI format. Conference Proceedings of the International Center for Scientific Research. Ternopil, Ukraine, September 19, 2025. P. 94–97. DOI: 10.62731/mcnd-19.09.2025.004.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/1053

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API – прикладний програмний інтерфейс (Application Programming Interface)

CASE – автоматизована система розробки програмного забезпечення (Computer-Aided Software Engineering)

CI/CD – безперервна інтеграція та безперервна доставка (Continuous Integration/Continuous Delivery)

CLI – інтерфейс командного рядка (Command Line Interface)

EA – Enterprise Architect

EMF – фреймворк моделювання Eclipse (Eclipse Modeling Framework)

ILP – цілочисельне лінійне програмування (Integer Linear Programming)

JSON – нотація об'єктів JavaScript (JavaScript Object Notation)

MDA – архітектура, керована моделлю (Model-Driven Architecture)

MDE – модельно-орієнтована інженерія (Model-Driven Engineering)

MOF – метаоб’єктна інфраструктура (Meta-Object Facility)

OCL – мова об'єктних обмежень (Object Constraint Language)

OMG – група управління об'єктами (Object Management Group)

PIM – платформно-незалежна модель (Platform-Independent Model)

PSM – платформно-специфічна модель (Platform-Specific Model)

RPC – віддалений виклик процедур (Remote Procedure Call)

SAT – задача здійсненності булевих формул (Boolean Satisfiability Problem)

SMT – задача здійсненності формул у теоріях (Satisfiability Modulo Theories)

TGG – потрійні графові граматики (Triple Graph Grammars)

UML – уніфікована мова моделювання (Unified Modeling Language)

VCS – система керування версіями (Version Control System)

XMI – обмін метаданими XML (XML Metadata Interchange)

YAML – YAML не є мовою розмітки (YAML Ain't Markup Language)

ВСТУП

Актуальність теми. Зростання складності та масштабу сучасних програмних систем ставить перед інженерією програмного забезпечення фундаментальне завдання управління архітектурною цілісністю протягом усього життєвого циклу. Ключову роль у вирішенні цієї проблеми відіграє модельно-орієнтована інженерія (model-driven engineering, MDE), що використовує мову UML (Unified Modeling Language) як стандартизований засіб для візуалізації, специфікації та документування архітектурних рішень. Проте ефективність MDE безпосередньо залежить від вибору формату зберігання моделей, який повинен задовольняти два суперечливі критерії: гнучкість для швидкої ітеративної розробки та формальну строгість для забезпечення достовірності.

Аналіз домінуючих форматів виявляє глибокий конфлікт. З одного боку, стандарт XMI (XML Metadata Interchange), розроблений Object Management Group (OMG), забезпечує повну відповідність специфікації UML, гарантуючи максимальну деталізацію та інтероперабельність між різними CASE-засобами. Це робить його незамінним для формального опису складної поведінкової семантики, такої як діаграми станів чи послідовностей. Водночас надлишковий синтаксис та громіздкість XMI-файлів ускладнюють їх ручний аналіз та створюють значні перешкоди для інтеграції з сучасними системами контролю версій, де навіть незначні зміни в моделі призводять до масштабних та неінформативних конфліктів злиття.

З іншого боку, формат JSON (JavaScript Object Notation) набув широкої популярності завдяки своїй простоті, компактності та легкості інтеграції в ітеративні процеси розробки. Він ідеально підходить для опису статичних структур (класів, атрибутів, пакетів), які часто змінюються, та забезпечує прозору роботу з системами контролю версій. Але при цьому ця гнучкість досягається ціною формалізму: JSON не має офіційного стандарту для представлення UML та позбавлений вбудованих механізмів для опису складної поведінкової семантики й крос-посилань між елементами, що є сильною стороною XMI.

В рамках дослідження був проведений аналіз сучасних CASE-засобів, таких як

StarUML, Modelio, Enterprise Architect, Visual Paradigm та MagicDraw/Cameo, який показує, що жоден з них не вирішує цю дилему комплексно. Інструменти змушують розробників робити компромісний вибір: або гнучкість та зручність JSON з обмеженим формалізмом, або строгість XMI, що ускладнює динамічну розробку, або використання закритих пропрієтарних форматів, які обмежують інтероперабельність.

Цей розрив між гнучкістю та формалізмом призводить до серйозних негативних наслідків. Він провокує так званий архітектурний дрейф – тобто поступове накопичення розбіжностей між проектною моделлю та її реальною імплементацією. Як наслідок, це призводить до виникнення прихованих архітектурних дефектів: семантичні помилки, такі як невідповідність сигнатур методів, «висячі» посилання на видалені елементи або використання застарілих типів даних. Такі дефекти не виявляються стандартними компіляторами і проявляються на пізніх етапах життєвого циклу – під час інтеграційного тестування або експлуатації, коли вартість їх виправлення зростає експоненційно. Все це призводить до того, що проблема забезпечення консистентності (тут і надалі в роботі терміни «консистентність» та «узгодженість» вживаються як синоніми) є не лише технічним питанням якості, а й ключовим фактором економічної ефективності розробки.

Отже, існує гостра науково-прикладна потреба в розробці нового науково-методичного апарату та інструментальних засобів, які б усунули існуючий компроміс. Необхідно створити підхід, що поєднує гнучкість JSON для опису статичної структури та формальну строгість XMI для поведінки в рамках єдиної, цілісної та гарантовано узгодженої моделі, забезпечуючи її ефективну перевірку в умовах сучасної ітеративної розробки.

Зв’язок роботи з науковими програмами, планами, темами. Чинну дисертаційну роботу виконано у відповідності до пріоритетних напрямків науково-дослідних робіт Національного університету «Одеська політехніка», згідно координаційних планів Міністерства освіти і науки України, зокрема, в рамках наукових досліджень за держбюджетними науково-дослідними роботами (НДР): «Методи та програмні засоби інтерпретації моделей машинного навчання непараметричних динамічних об’єктів» №210-63, 2021 – 2025 р. р. (№ держ.

реєстрації 0122U002161) та «Методи та технології моделювання і забезпечення якості складних програмних систем у динамічних середовищах» №264-73, 2025 – 2027 р. р. (№ держ. реєстрації 0125U003655).

Метою роботи є підвищення ефективності та достовірності процесу модельно-орієнтованої розробки складних програмних систем шляхом розробки науково-методичного апарату та інструментальних засобів для забезпечення гарантованої консистентності UML-моделей, що складаються із структурного подання у форматі JSON та поведінкового подання форматі XMI.

Під ефективністю мається на увазі сукупність показників, що характеризують ресурсоємність процесу валідації. Ключовими метриками цього процесу є час виконання та обсяг використаної оперативної пам'яті.

Достовірність – це здатність методу гарантовано виявляти семантичні помилки неузгодженості (на основі розробленої таксономії) та підтримувати модель у консистентному стані.

Для досягнення поставленої мети було сформульовано та вирішено такі **задачі**:

– Проведено аналіз проблеми забезпечення консистентності UML-моделей, існуючих форматів їх зберігання та функціональності сучасних CASE-засобів для виявлення їх існуючих недоліків та обґрунтування комбінованого підходу зі структурним та поведінковим поданнями. Додатково було сформульовано формальну класифікацію помилок неузгодженості між структурними та поведінковими поданнями для обґрунтування актуальності проблеми.

– Розроблено формальну метамодель UML з двома поданнями, що математично описує структурне та поведінкове подання, та встановлено відношення відповідності між ними. Для структурного подання було сформовано узагальнений підхід до його відображення, ідентифікації елементів та дворівневої валідації у JSON; для поведінкового подання так само були визначені та описані узагальнені правила його представлення через профіль UML, трасування до структури та перевірки формальних інваріантів у XMI. Додатково, для цієї метамоделі було сформульовано набір правил консистентності мовою OCL та обґрунтовано їх трансформацію до формалізму, придатного для автоматизованої верифікації за допомогою Alloy.

– Розроблено метод автоматизованого забезпечення консистентності, який базується на комбінованому застосуванні OCL-інваріантів для перевірки правил та трансформації до формалізму Alloy для глибокої верифікації. Сформульовано та доведено теорему інкрементальної консистентності, яка теоретично обґрунтовує метод інкрементальної валідації на основі графа залежностей з класифікованими ребрами. Також розроблено метод двосторонньої синхронізації та алгоритми трансформації між монолітним поданням у форматі XMI та запропонованою моделлю з двома поданнями для забезпечення сумісності з існуючими інструментами.

– Розроблено архітектуру та реалізовано прототип програмного плагіна, що втілює запропоновані методи, та проведено експериментальну оцінку його ефективності й продуктивності у порівнянні з існуючими промисловими аналогами.

Об'єкт дослідження – процес модельно-орієнтованої розробки програмного забезпечення.

Предмет дослідження – методи, моделі та інструментальні засоби забезпечення консистентності UML-моделей з двома поданнями.

Методи дослідження. Для досягнення поставленої мети у роботі використано комплекс наукових методів: системний аналіз для дослідження проблеми узгодженості, аналізу форматів зберігання та класифікації CASE-засобів; теорія метамоделювання та стандарт MOF для розробки формальної метамоделі UML з двома поданнями; теорія формальних мов і логік (OCL, SAT-аналіз) для визначення інваріантів узгодженості та підходу до їх автоматизованої верифікації; теорія графів для побудови графа залежностей і алгоритмів інкрементальної валідації; об’єктно-орієнтоване проектування та програмування для архітектури і реалізації плагіна; методи експериментальних досліджень і порівняльного аналізу для оцінки продуктивності і порівняння з аналогами.

Наукова новизна одержаних результатів, які виносяться на захист, полягає у наступному:

– *Вперше розроблено* формальну метамодель UML, яка охоплює структурне та поведінкове подання, реалізовані на основі комбінованого використання форматів JSON та XMI відповідно, які логічно пов’язані між собою через формально визначене

відношення відповідності; для цих подань сформульовано та узагальнено підходи до їх формалізації (для JSON) та правил представлення й валідації (для XMI). На відміну від існуючих підходів, що використовують монолітні або пропрієтарні формати, запропонована метамодель одночасно забезпечує компактне та VCS-дружнє зберігання статичної структури, адаптовану до частих і швидких змін, та підтримку стандартного XMI-представлення для динамічних аспектів, що забезпечує як сумісність із існуючими інструментами моделювання, так і ефективну інтеграцію в сучасні ітеративні процеси розробки.

– *Вперше запропоновано* метод автоматизованого забезпечення консистентності UML-моделі, побудованої за принципом двох подань. Метод функціонує на основі формально визначеного відношення відповідності між структурними та поведінковими елементами та базується на застосуванні OCL-інваріантів для верифікації локальної та глобальної узгодженості моделей при інкрементальному редагуванні. Для подолання обмежень OCL, він також включає двоступеневий підхід з трансформацією моделі у специфікацію Alloy для глибокої верифікації за допомогою SAT-вирішувачів. На відміну від відомих методів валідації, запропонований метод дозволяє проводити глибоку формальну верифікацію складних транзитивних властивостей та автоматично генерувати контрприклади для дефектів, що підвищує точність і повноту виявлення семантичних неузгодженостей.

– *Вперше сформульовано та доведено* теорему інкрементальної консистентності, яка гарантує, що будь-яка послідовність допустимих змін над структурним або поведінковим поданням, за умови виконання локалізованих перевірок OCL-інваріантів, зберігає глобальну узгодженість моделі в межах формального відношення відповідності. На відміну від традиційних підходів повної перевірки (зі складністю O(N)), це дозволяє теоретично обґрунтувати коректність та обчислювальну ефективність (зі складністю O(k·d)) інкрементальних методів валідації, завдяки чому можливе створення інструментів, придатних для інтеграції в CI/CD та IDE.

– *Отримав подальший розвиток* метод двосторонньої синхронізації, що реалізований через механізм інкрементальних оновлень та локалізованої перевірки

консистентності, яка ґрунтується на формальному графі залежностей із запропонованою класифікацією його ребер на структурні, семантичні та трасувальні. Особливістю розробленого методу є поєднання автоматичного виявлення конфліктів, формалізованих правил трансформації моделей і підтримки взаємної актуальності обох подань у реальному часі. На відміну від існуючих підходів, що часто є односпрямованими або покладаються на ручне втручання, запропонований метод забезпечує проактивну підтримку узгодженості в обох напрямках, що дозволяє одночасно вносити зміни як з боку коду, так і з боку моделі, запобігаючи архітектурному дрейфу.

Практичне значення одержаних результатів полягає у розробці конкретних методик, алгоритмів та інструментальних засобів, що можуть бути безпосередньо впроваджені в процеси розробки програмного забезпечення для підвищення їх ефективності та достовірності.

Розроблено та реалізовано прототип програмного плагіна, який втілює запропоновані методи та алгоритми. Плагін здатний автоматично будувати модель з вихідного коду, виконувати інкрементальну валідацію та може бути інтегрований у сучасні процеси безперервної інтеграції та доставки (Continuous Integration / Continuous Delivery, CI/CD) через реалізований інтерфейс командного рядка (Command Line Interface, CLI), що дозволяє автоматизувати контроль архітектурної цілісності на кожному етапі розробки.

Розроблені методики та алгоритми (інкрементальної валідації, двосторонньої синхронізації, прямої та зворотної трансформації моделей) можуть бути використані розробниками CASE-інструментів для розширення функціональності існуючих та створення нових засобів моделювання, що підтримують комбіновані підходи та забезпечують більш тісний зв'язок між моделлю та кодом.

В результаті, запропонований підхід дозволяє суттєво знизити вартість та трудомісткість розробки за рахунок раннього автоматизованого виявлення прихованих архітектурних дефектів на етапі проектування. В свою чергу, це запобігає їх прояву на пізніх стадіях життєвого циклу, де вартість виправлення є значно вищою, що підвищує загальну економічну ефективність та керованість проектів.

Особистий внесок здобувача. Усі основні наукові результати, викладені в дисертаційній роботі, отримані автором особисто. У наукових працях, опублікованих у співавторстві, здобувачеві належать такі ідеї та результати: ідея та архітектура метамоделі з двома поданнями, розробка методу інкрементальної валідації на основі графа залежностей, розробка спеціалізованого UML-профілю для трасування, формулювання та доведення теореми інкрементальної консистентності, а також проведення експериментальної перевірки ефективності запропонованих підходів.

Апробація результатів дисертації. Основні положення та результати дисертаційної роботи доповідались та обговорювались на 6 міжнародних наукових та науково-практичних конференціях: X міжнародна науково-практична конференція «Інформатика. Культура. Техніка» (Одеса, 2024); VIII міжнародна наукова конференція «Традиційні та інноваційні підходи до наукових досліджень» (Дрогобич, 2025); IX Міжнародна наукова конференція «Наукові тренди постіндустріального суспільства» (Тернопіль, 2025); IV Міжнародна наукова конференція «Технології та суспільство: взаємодія, вплив, трансформація» (Чернігів, 2025); X Міжнародна наукова конференція «Наукові тренди постіндустріального суспільства» (Рівне, 2025); V Міжнародна наукова конференція «Розвиток наук в умовах нової реальності: проблеми та перспективи» (Тернопіль, 2025).

Публікації. Основні наукові результати дисертації опубліковано у 12 наукових працях, у тому числі: 6 статей – в наукових фахових виданнях України категорії «Б», 6 публікацій у працях і матеріалах міжнародних наукових конференцій, 8 надруковано без співавторів.

Структура та обсяг роботи. Дисертація складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг дисертації становить 232 сторінки, з них основного тексту 191 сторінок. Робота містить 9 рисунків, 18 таблиць. Список використаних джерел налічує 167 найменувань.

1 АНАЛІЗ ПРОБЛЕМИ ЗАБЕЗПЕЧЕННЯ КОНСИСТЕНТНОСТІ UML-МОДЕЛЕЙ В УМОВАХ КОМБІНОВАНОГО ЗБЕРІГАННЯ ДАНИХ

Сучасна інженерія програмного забезпечення характеризується переходом від коду-орієнтованих підходів до модельно-орієнтованих, у яких центральним артефактом розробки виступає архітектурна модель [119]. Основна перевага такого підходу полягає у зниженні складності програмних систем шляхом підвищення рівня абстракції [31, 134]. Модель виконує роль формалізованого контракту, що визначає структуру, поведінку та обмеження системи, і слугує основою для аналізу, генерації коду [33, 86, 99, 121] та автоматизованої верифікації [4, 28, 45].

Водночас виникає суперечність між вимогами до строгої, внутрішньо узгодженої моделі та практикою сучасної розробки. З одного боку, для виконання функції архітектурного контракту модель повинна бути консистентною [6, 8, 90], тобто позбавленою логічних протиріч між різними поданнями, зокрема між статичною структурою та динамічною поведінкою системи [1, 9, 87]. З іншого боку, гнучкі методології та процеси безперервної інтеграції висувають вимогу до моделі бути динамічним, легко модифікованим артефактом, що еволюціонує синхронно з кодом і вимогами [12, 42, 84, 151].

Спроби поєднати ці два підходи виявляють низку проблем. Формалізація і строгість моделей призвели до створення складних і громіздких форматів зберігання [51, 111], які ускладнюють інтеграцію з сучасними робочими процесами. Натомість адаптація до вимог швидкої розробки часто супроводжується втратою формальної строгості, що унеможливлює або значно ускладнює перевірку консистентності [57, 58].

Ігнорування проблеми консистентності має відчутні наслідки. Воно спричиняє виникнення семантичних неузгодженостей між моделлю та реалізацією, які накопичуються у вигляді архітектурного дрейфу та технічного боргу [13, 41, 105, 124]. Усунення таких дефектів на пізніх етапах життєвого циклу програмного забезпечення потребує значних ресурсів [24, 36, 52, 68], що робить управління консистентністю як технічно, так і економічно значущим завданням [50].

Для комплексного аналізу цієї проблематики розділ структуровано наступним чином. Спочатку розглянуто роль і розвиток мови UML як основного інструменту модельно-орієнтованої інженерії [2]. Далі наведено формальні визначення консистентності та її класифікацію [7, 113, 118]. Наступним кроком є порівняльний аналіз домінуючих форматів зберігання моделей, який ілюструє баланс між гнучкістю та формалізмом [108, 155]. Потім подано огляд сучасних CASE-засобів з акцентом на їх архітектурних обмеженнях. Завершує розділ таксономія типових помилок консистентності [89, 136, 137], що стане основою для розробки методичного апарату у подальших частинах дослідження.

1.1 Роль та еволюція UML в модельно-орієнтованій інженерії

MDE стала відповіддю на кризу складності у програмній інженерії кінця XX століття [119]. Вона запропонувала зсув парадигми від коду до моделей, перетворивши останні з артефактів документації на центральні елементи розробки [26].

Основна ідея MDE – підвищення рівня абстракції від платформно-залежного коду до платформно-незалежних моделей (Platform-Independent Model, PIM). Ці моделі фіксують знання про предметну область та архітектуру, ігноруючи технологічні деталі. Подальша розробка автоматизується через модельні трансформації, що перетворюють PIM на платформно-специфічні моделі (Platform-Specific Model , PSM) і, врешті-решт, на виконуваний код. Цей підхід, що описується формулою «Моделі + Трансформації = Програмне забезпечення», максимізує сумісність та спрощує проектування [22, 62, 115].

Таким чином, фокус розробки зміщується з імперативного кодування на декларативне створення точних і повних моделей. Теоретичне підґрунтя MDE заклали праці Брамбілли, Кабота, Віммера, Шмідта та Безивіна [21, 26, 119]. Практичною реалізацією цих принципів стала ініціатива архітектури, керованої моделлю (Model-Driven Architecture, MDA) від OMG, яка стандартизувала підхід на основі UML, Meta-Object Facility (MOF) та XMI [72, 139, 155].

UML утвердилася як де-факто стандарт для MDE. Вона виникла внаслідок об'єднання методів Буча, Рамбо та Якобсона, що поклало край «війнам методів» 1990-х років і створило єдину мову (lingua franca) для розробників [141]. Консорціум OMG стандартизував UML у 1997 році та досі керує його розвитком [143].

UML виконує функції візуалізації, специфікації, конструювання та документування [140]. Проте в контексті MDE ключовою є функція специфікації, адже модель повинна бути не просто ілюстрацією, а формальним описом системи, придатним для автоматизованої трансформації.

Головною перевагою UML є багатий набір діаграм для комплексного моделювання системи (рис. 1.1).

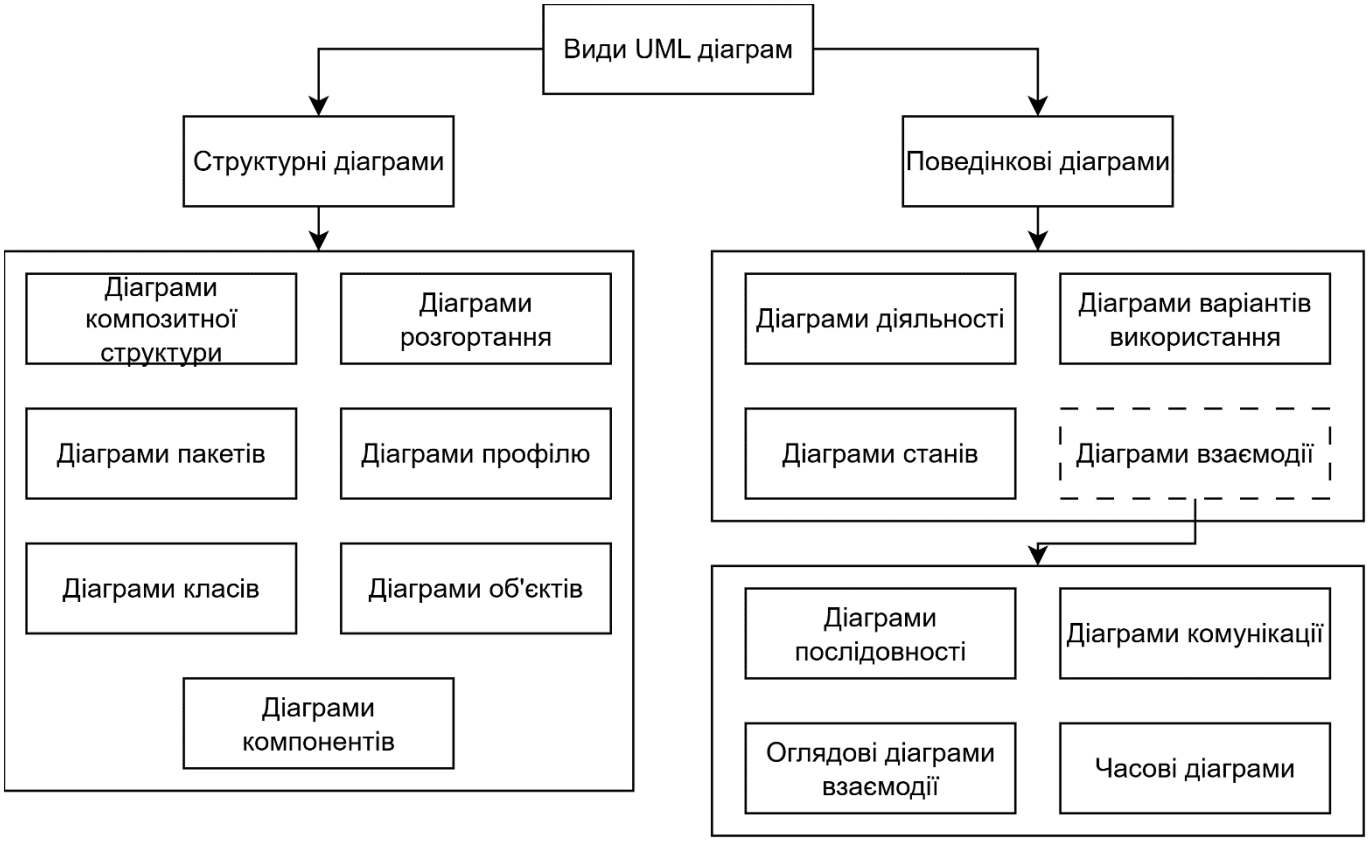

Рисунок 1.1 – Типи UML діаграм

Структурні діаграми (наприклад, діаграми класів, компонентів, розгортання) описують статичну структуру системи, її елементи та відношення між ними. Поведінкові діаграми (наприклад, діаграми діяльності, станів, варіантів використання) моделюють динамічні аспекти системи, її поведінку та реакцію на

події. Діаграми взаємодії (наприклад, діаграми послідовності, комунікації), що є підмножиною поведінкових, фокусуються на потоках керування та даних між об'єктами [16].

Саме ця здатність до багатоаспектного опису робить UML незамінним інструментом для створення повних моделей [143].

Однак ранні версії стандарту (UML 1.x), хоч і уніфікували нотації, мали недостатні виразні можливості для цілей MDE. Їхні обмеження в моделюванні складної поведінки та архітектури робили їх радше засобом документування, аніж основою для автоматичної генерації коду [5, 57, 73].

Прийняття стандарту UML 2.x у 2003–2005 роках стало стрибком, що перетворив мову з інструменту дизайну на засіб моделювання виконуваної архітектури [2, 20, 139]. Ключові нововведення кардинально розширили її виразну потужність. Зокрема, було покращено моделювання поведінки: діаграми діяльності отримали формальну семантику, подібну до мереж Петрі [97, 116], для точного опису складних потоків керування, а діаграми послідовності збагатились фрагментами взаємодії для моделювання нетривіальних сценаріїв [130]. Одночасно, введення структурованих класифікаторів, частин та портів забезпечило можливість архітектурної декомпозиції, що є фундаментальним для компонентно-орієнтованого проектування [2, 96, 139].

Ці вдосконалення перетворили UML на потужну мову для специфікації, придатну для автоматизованих трансформацій [119]. Однак ціною цього стало значне ускладнення метамоделі, що зробило її стандартний формат серіалізації, XMI, надзвичайно деталізованим, надлишковим і важким для обробки [72, 133, 155]. Навіть семантично неважливі зміни в моделі, як зміна координат елемента, призводили до значних модифікацій у XMI-файлі [149]. Це створило серйозні проблеми для інтеграції з сучасними системами контролю версій, такими як Git, де аналіз змін та злиття гілок стали вкрай ускладненими через велику кількість «шуму» [27, 53, 120]. В результаті виник парадокс: еволюція UML, спрямована на вирішення проблеми складності проектування, породила нову складність на рівні інструментальної підтримки [120].

Прагнення вирішити проблему складності форматів зберігання, що виникла внаслідок еволюції UML, підводить до аналізу фундаментальних принципів програмної архітектури. Ідея розгляду єдиної моделі через окремі, взаємопов'язані подання не є новою, тому доцільно проаналізувати впливові підходи – класичну модель «4+1» та сучасну C4 – щоб виявити їхні обмеження, які створюють передумови для проблем з консистентністю [83].

Модель «4+1» (рис. 1.2) пропонує описувати архітектуру програмно-інтенсивної системи з п'яти різних, але взаємопов'язаних подань, кожна з яких призначена для певного кола учасників даного процесу та вирішує специфічні задачі [83]:

1. *Логічне подання***.** Описує функціональність, яку система надає кінцевим користувачам. Фокусується на статичній структурі системи – класах, їхніх атрибутах, операціях та відношеннях.

2. *Подання процесів***.** Описує динамічні аспекти системи – процеси, потоки керування, взаємодію, паралелізм та синхронізацію. Цей погляд стосується нефункціональних вимог, таких як продуктивність та масштабованість.

3. *Подання розробки.* Описує статичну організацію програмного забезпечення в середовищі розробки (пакети, модулі, компоненти).

4. *Фізичне подання.* Описує відображення програмних компонентів на апаратні вузли.

5. *Подання сценаріїв.* Ілюструють та валідують архітектуру, описуючи послідовності взаємодій між компонентами для реалізації ключових варіантів використання.

Аналіз моделі виявляє фундаментальну дихотомію між логічним поданням та поданням процесів. Вони описують аспекти системи з діаметрально протилежними вимогами до їх фізичного представлення. Логічне подання, хоч і фіксує статичну сутність системи, є найбільш мінливою частиною моделі в процесі розробки. Воно постійно еволюціонує в ітеративних циклах, вимагаючи гнучкого формату зберігання, сумісного з системами контролю версій. Навпаки, подання процесів, що описує динаміку, зазвичай є архітектурно стабільним, але вимагає максимальної

формальної точності для коректного опису складних взаємодій. Цей концептуальний розрив створює архітектурний глухий кут на рівні інструментальної реалізації. Існуючі CASE-засоби ігнорують цю дихотомію, змушуючи розробників зберігати обидва подання в єдиному монолітному форматі, що породжує проблеми з версіонуванням та підтримкою моделей [55, 107, 153].

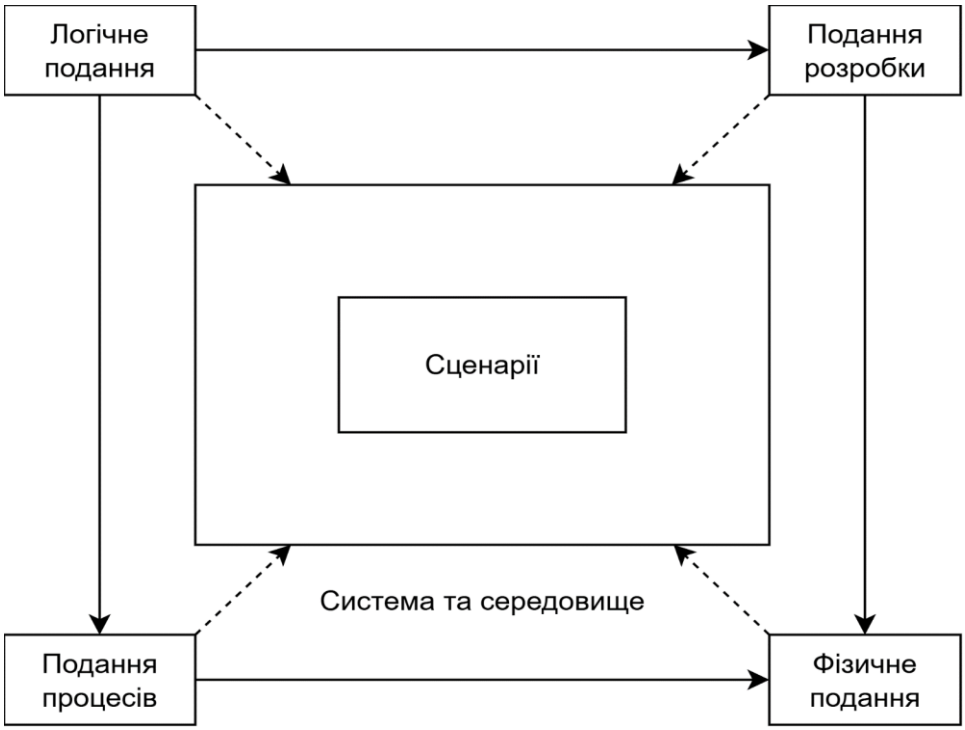

Рисунок 1.2 – Модель «4+1»

Як відповідь на надмірну складність та формалізм UML, з'явилися більш прагматичні підходи. Модель C4 (Context, Containers, Components, and Code), розроблена Саймоном Брауном, фокусується на простоті та ефективності комунікації архітектурних рішень у команді розробки [58, 154].

Вона пропонує описувати статичну структуру системи на чотирьох рівнях деталізації:

1. *Контекст.* Найвищий рівень, що показує систему як «чорну скриньку» в її оточенні – користувачів та зовнішні системи.

2. *Контейнери.* Декомпозиція системи на окремі одиниці розгортання (веб-застосунки, мікросервіси, бази даних).

3. *Компоненти.* Декомпозиція контейнера на логічні складові (наприклад, контролери, сервіси, репозиторії).

4. *Код.* Найнижчий рівень, що за потреби може деталізувати реалізацію окремих компонентів до рівня класів.

Схематична дана модель зображена на рис. 1.3

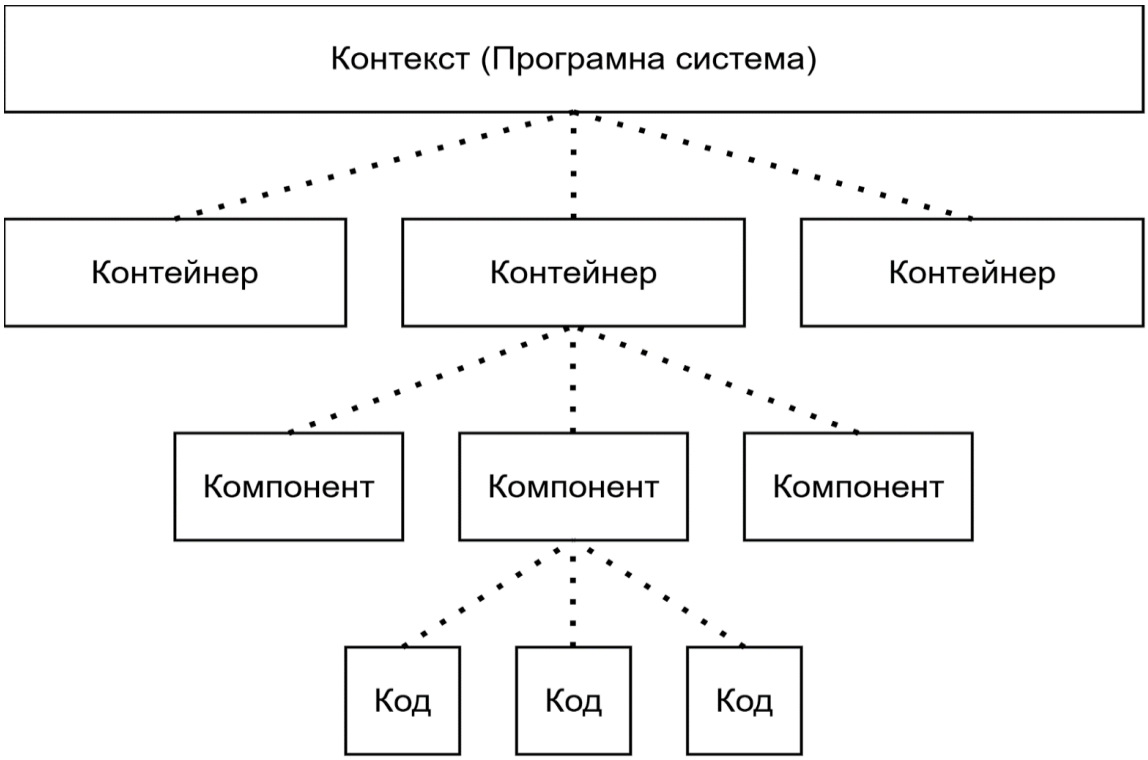

Рисунок 1.3 – Модель C4

Модель C4 ефективна для візуалізації статичної архітектури завдяки своїй простоті та чіткій ієрархії, що дозволяє створювати зрозумілі та легкі в підтримці діаграми. Однак модель C4 навмисно уникає детального моделювання динамічної поведінки. Для опису складних взаємодій вона пропонує використовувати UML-діаграми (послідовності, станів) як доповнення, але не вирішує проблем їхнього зберігання, версіонування та, найголовніше, забезпечення консистентності з основною структурною моделлю [50].

Аналіз обох підходів виявляє прогалину в існуючих архітектурних практиках. Класична модель «4+1» у її інструментальній реалізації є надто жорсткою та монолітною, створюючи конфлікт між гнучкістю та формалізмом [83]. Натомість

модель C4 є гнучкою для опису структури, але семантично недостатньою для моделювання складної поведінки [90].

Жоден з цих підходів не пропонує комплексного рішення, яке б поєднувало гнучкість опису статичної структури з формальною строгістю для верифікації динаміки. Саме ця відсутність збалансованого підходу до фізичного представлення моделі є ключовою першопричиною неузгодженостей та архітектурного дрейфу [72, 149]. Це обґрунтовує науково-практичну потребу в розробці нового методу, який би усунув цей компроміс [21, 119].

1.2 Проблема забезпечення консистентності в MDE

Як було показано в попередньому підрозділі, еволюція стандарту UML 2.x кардинально розширила його виразні можливості, перетворивши мову з інструменту візуалізації на потужний засіб для специфікації виконуваної архітектури [20, 57, 139]. Це дозволило моделювати складні статичні та динамічні аспекти програмних систем з високою деталізацією. Однак це досягнення не було безкоштовним. Ускладнення метамоделі та розповсюдження численних, тісно взаємопов’язаних діаграм неминуче породили нову фундаментальну проблему – проблему забезпечення консистентності між різними поданнями єдиної системної моделі [12, 17, 79].

Проблема консистентності є емерджентною властивістю, що виникає при моделюванні будь-якої складної системи. Парадигма MDE для управління складністю використовує принцип «розділяй і володарюй», декомпозуючи опис системи на набір спеціалізованих подань (структура, поведінка, розгортання) [119]. Ці подання не ізольовані, а описують ту саму систему з різних точок зору і містять інформацію, що перетинається. Наприклад, операція, визначена в діаграмі класів, може викликатись у діаграмі послідовності [136]. Саме цей перетин є джерелом потенційних суперечностей, оскільки зміна в одному поданні може вимагати узгоджених модифікацій в інших для збереження цілісності моделі [90]. Таким чином, сама стратегія управління складністю породжує проблему другого порядку – необхідність управління консистентністю [50].

В контексті MDE поняття «консистентність» виходить за межі простої синтаксичної коректності. Модель може відповідати правилам метамоделі UML, але містити глибокі логічні суперечності [7, 113]. Наприклад, діаграма послідовності може моделювати синтаксично допустимий, але семантично некоректний виклик приватного методу об'єкта ззовні, що порушує принципи інкапсуляції, визначені в діаграмі класів [2, 136].

Формально, багатоаспектна модель є консистентною, якщо вона *спільно реалізовна*. Це означає, що повинна існувати хоча б одна гіпотетична програмна система, яка б одночасно задовольняла всім твердженням, вираженим у всіх діаграмах моделі [39]. Іншими словами, консистентна модель не містить внутрішніх логічних суперечностей і описує реалізовний стан речей. Якщо ж така система не може існувати через наявність взаємовиключних тверджень, модель вважається неконсистентною і, отже, нереалізовною.

Ця семантична цілісність гарантується не лише базовими правилами UML, а й набором строгих формальних правил, відомих як інваріанти. Ці інваріанти, що часто виражаються декларативними мовами, наприклад, Object Constraint Language (OCL), фіксують обмеження, що мають виконуватися для будь-якого екземпляра моделі [32, 47, 66, 146, 156]. Наприклад, інваріант може вимагати, щоб сигнатура операції в діаграмі класів відповідала сигнатурі її виклику в діаграмі послідовності [135].

Отже, прагнення до консистентності перетворює моделювання з неформальної діяльності на строгу інженерну дисципліну. Коли модель стає формальним, верифікованим артефактом, вона долає семантичний розрив між проектом та реалізацією і слугує надійною основою для автоматизованих процесів, таких як аналіз та генерація коду [28]. Саме це є центральною обіцянкою MDE [26].

Для системного аналізу багатогранної проблеми забезпечення консистентності її прояви класифікують за різними критеріями. Дослідники виділяють кілька ортогональних вимірів для структурування та локалізації неузгодженостей [7, 90].

За рівнем абстракції розрізняють *горизонтальну* та *вертикальну* консистентність. Горизонтальна (також відома як внутрішньомодельна) стосується узгодженості між поданнями на одному рівні абстракції, наприклад, між діаграмами

класів та послідовності [79]. В свою чергу вертикальна (або міжмодельна) забезпечує відповідність між моделями на різних рівнях, гарантуючи, що проектні рішення реалізують поставлені вимоги [113].

За природою правил виділяють *синтаксичну* та *семантичну* консистентність. Синтаксична означає відповідність моделі правилам метамоделі UML і зазвичай перевіряється автоматично. Семантична стосується логічної несуперечливості змісту моделі; неузгодженості такого типу є найбільш складними для виявлення та небезпечними за своїми наслідками [7].

Узагальнену класифікацію з прикладами типових порушень для кожного виду консистентності наведено в табл. 1.1.

Таблиця 1.1 – Класифікація видів консистентності моделей

Вимір	Тип консистентності	Визначення	Приклад порушення в UML
Рівень абстракції	Горизонтальна	Узгодженість між поданнями на одному рівні абстракції.	Повідомлення в діаграмі послідовності викликає операцію processOrder() для об'єкта класу Order, але в діаграмі класів цей клас не має такої операції.
	Вертикальна	Узгодженість між моделями на різних рівнях абстракції.	Діаграма варіантів використання визначає, що актор «Клієнт» може «Перевірити статус замовлення», але в проектній діаграмі класів відсутні методи для реалізації цієї функціональності.
Природа правил	Синтаксична	Відповідність моделі правилам метамоделі UML.	Відношення композиції (чорний ромб) вказує на клас, який є одночасно і частиною, і цілим, створюючи синтаксично некоректний цикл.
	Семантична	Логічна несуперечливість та сумісність поведінки.	Діаграма станів для об'єкта Замовлення показує перехід зі стану «Оплачено» у стан «Скасовано», але діаграма діяльності для процесу скасування вимагає, щоб замовлення було у стані «Очікує оплати».

Дане дисертаційне дослідження фокусується переважно на проблемі горизонтальної семантичної консистентності між структурним (діаграми класів) та поведінковим (діаграми послідовностей, станів, діяльності) поданнями в межах єдиної проектної моделі. Цей вибір обґрунтований тим, що саме на цьому рівні виникає найбільша кількість прихованих, критичних дефектів, які важко виявити автоматизованими засобами, і які мають значний вплив на якість кінцевого програмного продукту [79, 136].

Неконсистентність моделі є першопричиною серйозних патологій у розробці програмного забезпечення з суттєвими технічними та економічними наслідками. Порушення узгодженості створює семантичний розрив, через який проектна модель перестає бути достовірним джерелом інформації [119]. В результаті розробники не можуть покладатися на модель як на точний план, що змушує їх кодувати на основі застарілих уявлень [26].

Цей розрив є головним каталізатором архітектурного дрейфу – поступового відхилення реальної архітектури коду від початкового проекту [6]. Коли модель не підтримується в узгодженому стані, кожна зміна в коді, зроблена без її оновлення, поглиблює розбіжність. З часом система перетворюється на складне переплетення нескоординованих модифікацій, що значно ускладнює її розуміння та подальший розвиток.

Наслідком архітектурного дрейфу є накопичення прихованих архітектурних дефектів. Підступність полягає в тому, що це не просто поверхневі помилки кодування, а глибокі семантичні помилки в структурі та логіці системи. До таких дефектів належать невідповідності сигнатур методів, висячі посилання на видалені класи та використання застарілих типів даних [137].

Сукупність цих проблем формує архітектурний технічний борг. Подібно до фінансового боргу, ігнорування неконсистентності дає короткостроковий виграш у швидкості, але в довгостроковій перспективі накопичує «відсотки» у вигляді значно збільшених витрат на супровід та рефакторинг [124, 126].

Критичність цієї проблеми підтверджується емпіричними дослідженнями, що демонструють експоненційне зростання вартості виправлення дефектів. Класичні

роботи Б. Боема, NASA та IBM показують, що дефект, виправлений на етапі експлуатації, коштує в 10-100 разів дорожче, ніж на етапі проектування [24, 52, 68]. Тому можна зробити висновок, що управління консистентністю є не питанням технічної досконалості, а ключовим фактором економічної ефективності проекту. Нехтування ним перетворює модель з цінного активу, що керує розробкою, на пасив, який генерує ризики та додаткові витрати [13, 142].

Аналіз доводить, що проблема забезпечення консистентності є центральною невирішеною задачею в сучасній модельно-орієнтованій інженерії [7, 50, 90]. Вона є прямим наслідком зростаючої складності систем та еволюції UML до багатоаспектного моделювання [57, 137], де цілісна система описується через набір взаємозалежних, але розрізнених діаграм. Консистентність – це глибока семантична властивість, що гарантує логічну цілісність моделі, а її порушення призводить до критичних наслідків: семантичних розривів, архітектурного дрейфу та накопичення технічного боргу, що має значний негативний економічний ефект [6].

Резюмуючи, формалізувавши критичність проблеми, необхідно проаналізувати її першоджерела на інструментальному рівні. Оскільки відсутність консистентності виникає через асинхронну еволюцію різних подань моделі, ключовим фактором є спосіб їхнього фізичного зберігання. Як буде показано далі, саме вибір формату є одним із головних джерел конфлікту між гнучкістю, якої вимагають ітеративні процеси, та формалізмом, необхідним для підтримки узгодженості [72].

1.3 Порівняльний аналіз форматів зберігання та обміну UML-моделями

Формалізувавши проблему консистентності як ключовий виклик для MDE, необхідно проаналізувати її першоджерела на технічному рівні. Вибір формату для зберігання та обміну UML-моделями є фундаментальним архітектурним рішенням, що безпосередньо впливає на консистентність, ефективність розробки та інтеграцію інженерних практик.

Домінуючі формати створюють фундаментальну напругу між двома критичними вимогами: потребою у формальній, машинооброблюваній строгості, яку

обіцяє MDE, та вимогою до гнучкої, людино-орієнтованої співпраці, що є основою підходів Agile та DevOps [119]. Цей підрозділ демонструє, як технічні характеристики існуючих форматів серіалізації стають епіцентром цього конфлікту, що безпосередньо призводить до описаних раніше проблем з консистентністю.

В першу чергу, мова буде йти про XMI та JSON.

Як вже було сказано раніше, формат XMI [155] є офіційним стандартом, розробленим та підтримуваним OMG, і слугує наріжним каменем ініціативи MDA [21]. Його основне призначення – забезпечити стандартизований механізм для обміну метаданими, зокрема UML-моделями, між різними CASE-засобами та репозиторіями в гетерогенних середовищах [72, 100]. Структура XMI не є довільною; вона формально генерується на основі метамоделі, визначеної за допомогою MOF, що гарантує повну та точну відповідність між абстрактними конструкціями UML та їхнім представленням у форматі XML [139].

Перш за все, необхідно описати його сильні сторони.

Головною перевагою XMI є його формальна строгість та семантична повнота. Завдяки прямому зв'язку з метамоделлю UML, XMI здатний представити всі нюанси специфікації UML 2.x, включаючи складні поведінкові діаграми, архітектурні патерни та розширення через профілі. Ключову роль у цьому відіграють механізми xmi:id та xmi:idref. Ці атрибути дозволяють серіалізувати складну, неієрархічну графову структуру UML-моделі у вигляді ієрархічного XML-документа. Кожен елемент моделі отримує унікальний ідентифікатор (xmi:id), на який можуть посилатися інші елементи (xmi:idref), що уможливлює представлення таких відношень, як асоціації, узагальнення, залежності та складні взаємозв'язки в діаграмах послідовності чи станів. Ця здатність до точного відтворення семантики робить XMI незамінним інструментом для задач, що вимагають формальної верифікації, аналізу моделей та автоматизованої генерації коду, що є центральною ідеєю MDE [21]. Завдяки стандартизації, XMI позиціонується як універсальна lingua franca, що має забезпечувати безшовну інтероперабельність між різними інструментами моделювання.

Попри формальні переваги, на практиці XMI демонструє низку суттєвих

недоліків, які ускладнюють його використання в сучасних ітеративних процесах розробки. В першу чергу, це надлишковість та низька читабельність для людини [108]. Оскільки XMI повинен детально описувати кожен аспект метамоделі, результуючі файли стають громіздкими, переповненими службовими тегами та атрибутами, що робить їх практично неможливими для ручного аналізу чи редагування.

Найкритичнішою проблемою XMI є його фундаментальна несумісність із принципами роботи розподілених систем контролю версій, таких як Git, як найбільш популярна система [149]. Ця несумісність випливає з глибокої технічної розбіжності між природою XMI-документів та алгоритмами контролю версій.

По-перше, *недетермінована серіалізація* є значною перешкодою. CASE-засоби часто зберігають елементи моделі в XMI-файлі у довільному порядку. Це означає, що повторне збереження моделі, навіть без логічних змін, може повністю переставити XML-блоки. Для лінійно-орієнтованого алгоритму «diff», який використовує Git, такий файл виглядає як повністю змінений, що генерує беззмістовні звіти про відмінності та робить аналіз історії неможливим.

По-друге, існує *невідповідність гранулярності*. Навіть локалізована зміна, така як перейменування атрибута, може спричинити правки в кількох віддалених рядках XMI-файлу. Алгоритми «diff» не здатні розпізнати семантичну єдність такої операції, представляючи її як набір непов'язаних видалень та додавань і приховуючи намір розробника.

По-третє, це призводить до *непрозорих конфліктів злиття*. При паралельній роботі спроба злиття змін майже неминуче викликає конфлікти [91]. Git представляє їх як перетин блоків сирого XML-тексту без семантичного контексту [27]. Вирішення таких конфліктів вручну вимагає глибокого розуміння структури XMI та є надзвичайно складним і схильним до помилок процесом.

Ці проблеми роблять ефективну спільну роботу над UML-моделями у форматі XMI з використанням стандартних робочих процесів Git практично неможливою. Команди змушені вдаватися до неефективних практик, наприклад ексклюзивне блокування файлів, що нівелює переваги розподіленої розробки. Непридатність XMI

для сучасних систем контролю версій (Version Control System, VCS) є наслідком його історичного походження: формат був розроблений в епоху до домінування Git та оптимізований для машинного обміну даними і формальної повноти, а не для зручності текстового версіонування.

Формат JSON, визначений стандартами ECMA-404 та RFC 8259, є легковагим текстовим стандартом обміну даними. На відміну від формалізму XMI, він базується на простому синтаксисі, що складається з об'єктів (колекцій пар «ключ-значення») та масивів (впорядкованих списків) [48].

Головними перевагами JSON є простота, продуктивність, висока читабельність для людини та легкість обробки практично будь-якою мовою програмування. Ці властивості роблять його зручним для VCS. Оскільки зміни в JSON-файлах зазвичай локалізовані та передбачувані, Git генерує зрозумілі звіти про відмінності і надійніше виконує автоматичне злиття гілок. Це зробило JSON де-факто стандартом для конфігураційних файлів та API, де пріоритетом є гнучкість та зручність для розробника [108, 123].

Однак гнучкість JSON досягається ціною формалізму та семантичної виразності, що є серйозною перешкодою для його використання в UML-моделюванні. Ключовий недолік – відсутність офіційного стандарту OMG для представлення UML-моделей у JSON. Це призвело до фрагментації ринку на численні пропрієтарні, несумісні між собою реалізації (наприклад, формат .mdj інструменту StarUML) [15, 127], що повністю підриває принцип інтероперабельності – наріжний камінь MDE.

Глибша технічна проблема полягає в ієрархічній природі JSON, яка не має вбудованих стандартних механізмів для неієрархічних зв'язків та перехресних посилань, необхідних для графової структури UML-моделі. Хоча існують механізми, як $ref у JSON Schema, їхнє застосування для складних UML-відношень є нестандартним і залежить від конкретної реалізації інструменту [67].

Це структурне обмеження робить JSON непридатним для точного опису складної семантики поведінкових діаграм (переходів у діаграмах станів, фрагментів взаємодії, потоків даних), що значною мірою покладаються на систему посилань. Спроба відтворити таку логіку в JSON призводить або до втрати інформації, або до

створення надскладних, нестандартних документів. Таким чином, простота формату, що є його перевагою, стає його головним недоліком у задачах формального моделювання [105, 108].

Отже, простота формату, що є його перевагою, стає його головним недоліком у задачах формального моделювання [155].

Аналіз XMI та JSON виявляє фундаментальний компроміс між формалізмом та гнучкістю. Розгляд ключових альтернатив доводить, що ця дилема є загальною для серіалізації даних і що жоден унітарний формат не забезпечує оптимального балансу для спільного UML-моделювання [48, 100].

YAML. Формат YAML (YAML Ain't Markup Language), будучи синтаксичною надмножиною JSON, пропонує кращу читабельність завдяки відступам та підтримці коментарів [76]. Однак, попри покращену ергономіку для розробника, він успадковує ключову слабкість JSON у контексті MDE: відсутність стандартизованого та семантично багатого відображення для UML. Це не вирішує проблем формальної верифікації та інтероперабельності моделей.

Бінарні формати. Бінарні формати, наприклад, Protocol Buffers та Avro оптимізовані для максимальної продуктивності та компактності, що робить їх ідеальними для мікросервісних комунікацій та зберігання великих обсягів даних [104]. Проте вони є непрозорими «чорними скриньками» для людини та, що більш важливо, текстово-орієнтованих систем контролю версій. Git відстежує такі файли як неподільні об'єкти, роблячи операції diff та merge неможливими. Це робить їх абсолютно непридатними для сучасних колаборативних процесів розробки [91].

Бази даних. Зберігання моделей у спеціалізованих базах даних – потужна альтернатива для керування великими моделями. Графові бази даних, наприклад, Neo4j, особливо ефективні, оскільки їхня структура природно відповідає графовій природі UML. Однак цей підхід створює архітектурний розрив із сучасною, файло-орієнтованою екосистемою розробки, яка включає процеси CI/CD та Git. Інтеграція такого рішення в стандартний конвеєр є складною і часто створює ізольоване «сховище моделей», відірване від інших артефактів проекту [80].

Аналіз альтернатив підтверджує, що універсального рішення для серіалізації

моделей не існує. Кожен формат є інженерним компромісом, що оптимізує одні характеристики за рахунок інших. Отже, проблема полягає не в пошуку єдиного «кращого» формату, а в розробці стратегії, яка дозволить інтелектуально комбінувати різні підходи.

Таким чином, проведений аналіз показує, що XMI та JSON є двома крайнощами на спектрі форматів серіалізації. Формалізм та семантична повнота XMI компенсують їх брак у JSON, тоді як гнучкість та швидкість JSON вирішують ключові проблеми XMI в сучасних інженерних практиках [48]. Ця взаємодоповнюваність обґрунтовує центральну гіпотезу дослідження: оптимальним рішенням є не вибір одного компромісного формату, а свідомий поділ відповідальності – використання кожного для тих завдань, де його переваги максимальні [149].

Концепція комбінованої моделі передбачає такий розподіл.

Статична структура (класи, атрибути, пакети), яка є найбільш динамічною і часто змінюється, зберігається у JSON. Це забезпечує легкість редагування, прозорість змін у системах контролю версій та надійне автоматичне злиття, що є критично важливим для ефективної командної роботи [123].

Динамічна поведінка (діаграми станів, послідовностей, діяльності), яка вимагає максимальної формальної точності, зберігається в XMI, що гарантує повну відповідність специфікації UML 2.x, можливість застосування формальних правил валідації та сумісність з існуючими інструментами аналізу та симуляції [37, 109, 139, 146, 155].

Такий підхід має глибоке архітектурне обґрунтування, оскільки відображає модель «4+1» Філіпа Крухтена: структурне подання в JSON реалізує Логічний погляд (що система є), а поведінкове в XMI – Процесний погляд (що система робить) [83]. Це є оптимальним інженерним компромісом, що узгоджує фізичну стратегію зберігання з концептуальними поданнями та вирішує конфлікт між парадигмами MDE та Agile/DevOps. Формалізація та інструментальна підтримка цього комбінованого підходу є предметом данних робіт [26].

Порівняльний аналіз доводить, що ані стандартний XMI, ані гнучкий JSON, ані їхні альтернативи не є універсальним рішенням. Технічні обмеження цих форматів –

зокрема конфлікт XMI з системами контролю версій та відсутність стандарту і семантичної повноти у JSON – є прямою причиною проблем з консистентністю моделей [105, 127, 128].

Проведений аналіз обґрунтовує, що найбільш перспективним напрямком є не пошук нового унітарного формату, а *комбінований підхід*, що базується на розділенні відповідальності. Зберігання статичної структури в JSON та динамічної поведінки в XMI дозволяє поєднати гнучкість ітеративної розробки з формальною строгістю. Цей підхід, підкріплений класичними принципами програмної архітектури, формує теоретичне підґрунтя для вирішення проблеми узгодженості [48, 72].

Обґрунтувавши доцільність даного підходу на рівні форматів даних, наступним кроком є аналіз того, як існуючі інструментальні засоби реалізують роботу з цими форматами та чи підтримують вони такий комбінований підхід [80, 92, 101].

1.4 Огляд та класифікація сучасних CASE-засобів

Після обґрунтування архітектурних переваг комбінованого підходу до зберігання UML-моделей, який поєднує гнучкість JSON для статичної структури та строгість XMI для динамічної поведінки, необхідно провести аналіз ринку сучасних засобів комп'ютеризованої інженерії програмного забезпечення. Мета – з’ясувати, чи реалізований подібний підхід у наявних промислових або академічних інструментах. Аналіз сфокусований не на переліку функціональних можливостей, а на виявленні фундаментальних архітектурних рішень та оцінці механізмів їх зберігання, валідації й інтеграції на предмет відповідності вимогам ітеративної розробки.

Для структурування аналізу застосовано набір ортогональних критеріїв. Вони слугують індикаторами конфлікту між двома парадигмами: модельно-орієнтованою інженерією [21], що вимагає формалізму, та гнучкими методологіями, які пріоритезують швидкість і адаптивність.

Критерії для аналізу та класифікації CASE-засобів описані нижче.

*Формат зберігання***.** Аналізуються пропрієтарність, текстова/бінарна природа та відповідність стандартам. Це визначає прозорість, інтероперабельність моделі та її

сумісність із системами контролю версій.

*Підтримка комбінованої моделі***.** Оцінюється нативна архітектурна підтримка одночасного використання різних форматів для єдиної концептуальної моделі, на противагу простій конвертації через імпорт/експорт.

*Тип валідації***.** Класифікація режимів роботи: повна (on-demand), що перевіряє всю модель; інкрементальна, що аналізує зміни та їхній вплив; та активна (real-time), що забезпечує безперервний зворотний зв'язок. Критерій визначає придатність інструменту для швидких ітераційних циклів.

*Механізм валідації***.** Досліджується основа перевірки: вбудовані жорстко закодовані правила, підтримка формальної мови обмежень та можливість кастомізації правил. Це визначає глибину та формальну строгість контролю консистентності.

Зв'язок «модель-код». Оцінюється рівень інтеграції з вихідним кодом – від односторонньої генерації до повноцінної двосторонньої синхронізації [86, 129].

*Інтеграція з CI/CD***.** Аналізується наявність та потужність інтерфейсу командного рядка та програмних інтерфейсів для автоматизації валідації у конвеєрах безперервної інтеграції та доставки [35, 69, 70, 71, 152].

Застосування цього аналітичного фреймворку дозволяє класифікувати архітектурні філософії інструментів, доводячи, що кожен з них демонструє сильний ухил в бік однієї з парадигм, що підтверджує відсутність збалансованого рішення на ринку.

Гетерогенний ринок CASE-засобів вимагає системного аналізу шляхом групування інструментів за їхньою домінантною філософією проектування, що відображає архітектурні компроміси в дилемі «гнучкість-формалізм».

Першу групу взяла за орієнтир саме гнучкість. До неї можна віднести JSON-орієнтований підхід інструменту StarUML. Він пріоритизує зручність та швидкість розробки, базуючи свою архітектуру на власному текстовому форматі .mdj – структурованому JSON-документі [127]. Таке рішення забезпечує переваги для сучасних інженерних практик: файли є людиночитабельними, легко піддаються скриптовій обробці та інтегруються із системами контролю версій на кшталт Git, генеруючи зрозумілі звіти про відмінності.

Проте ця гнучкість досягається ціною формалізму. Пропрієтарний формат .mdj створює залежність від постачальника та ускладнює інтероперабельність. Механізм валідації, попри наявність понад 50 вбудованих правил, обмежується синтаксичною коректністю та базовими правилами доброї сформованості в межах однієї діаграми. Відсутня глибока семантична перевірка узгодженості між різними поданнями, наприклад, між діаграмами класів та послідовності. Валідація є повною, а не інкрементальною чи активною, що знижує ефективність зворотного зв'язку [147].

Критичним недоліком є обмін даними через стандарт XMI. Реалізований як зовнішнє розширення, цей функціонал має задокументовані обмеження. Експорт в XMI 2.1 (на основі UML 2.0) призводить до втрати семантичної точності, а візуальне представлення діаграм не передається, оскільки це не є частиною стандарту [128]. Це позиціонує XMI як вторинний формат для обміну, а не як нативний засіб представлення моделі. Інтеграцію в CI/CD конвеєри ускладнює відсутність офіційного CLI для автоматизованої валідації [15].

Тому StarUML втілює філософію «модель як ескіз» – інструмент для швидкого прототипування, де формальна верифікованість є вторинною. Його архітектура свідомо жертвує строгістю MDE заради гнучкості, якої вимагають Agile-практики.

Друга група фокусуються на парадигмі стандартизації. Це EMF- та XMI-орієнтовані інструменти, такі як Modelio та Papyrus [80, 110]. На противагу гнучким підходам, інструменти на базі Eclipse Modeling Framework (EMF), як Papyrus, та інші стандарт-орієнтовані засоби, як Modelio, пріоритезують суворе дотримання стандартів OMG. Їхня архітектура базується на метамоделі UML та використовує XMI як основний формат серіалізації [155]. Papyrus розділяє модель на три файли (.uml, .notation, .di), а Modelio – на структуру каталогів з XMI-файлами [103].

Формально коректний за принципами MDE, цей підхід створює фундаментальні проблеми для сучасних інженерних практик. Складна, багатофайлова структура та надлишковість XMI роблять моделі практично некерованими в системах контролю версій [17, 91, 120]. Документація Modelio прямо застерігає від використання Git, оскільки злиття неминуче призводить до пошкодження моделі, що є визнанням несумісності архітектури з розподіленими

робочими процесами.

Сильною стороною цих інструментів є потужна валідація на основі OCL для перевірки складних семантичних обмежень [94, 146, 109]. Завдяки інтеграції з екосистемою Eclipse, Papyrus може використовувати сторонні фреймворки, як EMF-IncQuery, для інкрементальної валідації в реальному часі, хоча це вимагає значних зусиль з налаштування [18, 101]. Водночас відомим недоліком EMF-орієнтованих інструментів є низька продуктивність та високе споживання пам'яті на великих моделях [19, 81].

Отже, ці інструменти реалізують філософію «модель як формальний контракт», де пріоритетом є строгість та інтероперабельність. Проте їхня архітектура, успадкована з епохи централізованих процесів, створює бар'єр для інтеграції в гнучкі, файло-орієнтовані робочі процеси на основі Git [17, 91, 120].

Третя група бере за ціль пропрієтарну продуктивність в рамках бінарної та базо-орієнтованої архітектури. Enterprise Architect від Sparx Systems вирішує проблеми продуктивності та багатокористувацької роботи XMI-інструментів іншим шляхом. Його архітектура базується на пропрієтарних бінарних форматах на основі файлових баз даних (MS Access .eap, SQLite .qea) [98]. Це забезпечує високу швидкість доступу до елементів моделі та надійну одночасну роботу кількох користувачів [55].

Однак ця продуктивність досягається ціною втрати прозорості та інтероперабельності. Модель перетворюється на «чорну скриньку»[104], не доступну для аналізу стандартними текстовими інструментами. Для систем контролю версій, як Git, бінарний файл є неподільним об'єктом, що унеможливлює аналіз змін та злиття [91]. Експорт у XMI, хоч і можливий, часто призводить до втрати специфічної інформації та не є нативним для моделі.

Вбудовані механізми валідації підтримують перевірку правил UML та власних обмежень через OCL [4, 102]. Проте закритість моделі ускладнює їхню зовнішню автоматизацію, наприклад, у CI/CD конвеєрах. Інтеграція можлива лише через пропрієтарний API, що вимагає значних зусиль на розробку .

Таким чином, Enterprise Architect втілює філософію «модель як централізована база даних». Це потужне рішення для корпоративних середовищ, де пріоритетом є

спільна робота, але ціною є ізоляція від відкритої, файло-орієнтованої екосистеми розробки, яка домінує в сучасній інженерії [55, 100].

Четверта та остання група орієнтується на інтегровану валідацію. До неї відносяться передові комерційні платформи MagicDraw/Cameo [144] та Visual Paradigm [30]. Ці інструменти є ринковими лідерами завдяки потужним вбудованим механізмам валідації, що забезпечують консистентність моделей у режимі реального часу. MagicDraw/Cameo реалізує механізм активної валідації – фоновий процес, який безперервно аналізує модель, миттєво повідомляє про порушення та може бути розширений користувацькими OCL-правилами [3, 131]. Аналогічно, Visual Paradigm має вбудовані засоби, що попереджають про помилки безпосередньо в процесі моделювання [66].

Однак, попри переваги у валідації, архітектурно ці інструменти є традиційними. Вони використовують власні складні, пропрієтарні XML-подібні формати (.mdxml, .vpp) [92, 153]. Хоча ці формати текстові, їхня надлишкова структура створює ті ж проблеми для систем контролю версій, що й XMI, ускладнюючи аналіз змін та злиття [91, 155]. Наявні інтерфейси командного рядка призначені для керування самим інструментом (наприклад, для генерації звітів), а не для роботи з моделлю як з текстовим артефактом у CI/CD конвеєрах [35, 69].

Отже, ці платформи реалізують філософію «модель як інтелектуальне середовище», надаючи найкращий досвід для моделювальника. Вони ефективно вирішують проблему негайного зворотного зв'язку, але ціною створення ще більш закритих, складних та монолітних екосистем. Таким чином, усувається один симптом (повільна валідація), але поглиблюється першопричина – ізоляція моделювання від стандартних інженерних практик [3].

Синтез проведеного аналізу дозволяє сформулювати узагальнені висновки про фундаментальні недоліки, притаманні всьому ринку сучасних CASE-засобів. Ці недоліки не є проблемами окремих реалізацій, а свідчать про системну прогалину між можливостями інструментів та потребами сучасної інженерії програмного забезпечення [26, 134].

Відсутність нативної мультиформатної архітектури. Проведений аналіз

доводить, що всі наявні інструменти є архітектурно монолітними. Вони змушують робити компромісний вибір між гнучкістю, формалізмом та продуктивністю, обираючи єдиний домінуючий формат зберігання [15, 98, 155]. Жоден інструмент не побудований на ідеї нативного використання кількох оптимальних форматів для різних аспектів моделі як єдиного, цілісного артефакту. Функції імпорту та експорту є лише механізмами конвертації, а не основою архітектури.

Ця ситуація є наслідком архітектурної інерції, оскільки існуючі інструменти еволюціонували навколо єдиного механізму персистентності. Впровадження гібридної моделі вимагає не просто додавання парсера, а повного перепроектування ядра, що є дорогим та ризикованим через проблеми зворотної сумісності. Цей технологічний та економічний бар'єр створює стійку науково-практичну нішу. Оскільки наявні гравці ринку навряд чи будуть кардинально перебудовувати свої продукти, рішення, спроектоване з нуля на гібридній архітектурі, може запропонувати унікальну перевагу, важку для відтворення конкурентами [58].

Неефективність механізмів валідації для ітеративних процесів. Більшість інструментів покладаються на повну, «важку» валідацію за вимогою користувача. Навіть передова «активна валідація» (наприклад, у MagicDraw), хоч і ефективна для інтерактивної роботи, не призначена для миттєвого виконання в CI/CD конвеєрах, де швидкість є критичною [131]. Час повної перевірки великих моделей робить такі механізми непрактичними для автоматизованих процесів [51, 111].

Відсутність швидкої, інкрементальної валідації є однією з першопричин архітектурного дрейфу [6]. Цей феномен виникає, коли вартість підтримки моделі в узгодженому стані перевищує її цінність для розробника [142]. Повільний ручний запуск валідації призводить до того, що розробники уникають його, особливо під тиском дедлайнів. Це веде до накопичення неузгодженостей [7, 90], які переростають у значний архітектурний технічний борг [13, 82, 124]. Отже, ефективна інкрементальна валідація – це не просто бажана функція, а необхідна умова для підтримки моделі як живого та цінного артефакту впродовж усього життєвого циклу проекту [19, 34, 75, 81].

Хронічний розрив між моделлю та кодом. Незважаючи на десятиліття розвитку

MDE [21, 115], більшість CASE-засобів залишаються «островами моделювання», відірваними від вихідного коду. Механізми двосторонньої синхронізації є складними, ненадійними та вимагають значних ручних зусиль [58]. Жоден інструмент не реалізує підхід, де структурна модель автоматично генерується з актуального коду [129], а валідація перевіряє консистентність поведінкових діаграм із цією «живою» структурою .

Історично CASE-інструменти створювалися для архітекторів, що працюють «над» кодом. Сучасна культура DevOps стирає цю межу, але інструменти не еволюціонували відповідно і залишаються орієнтованими на моделювальників, а не на команди розробки [134]. Підхід генерації моделі з коду змінює цю парадигму: він інтегрує моделювання в щоденний процес розробки, роблячи модель релевантною для всієї команди [26].

Отже, проведений аналіз ринку CASE-засобів емпірично підтверджує існування значної науково-практичної прогалини [134]. Встановлено, що жоден інструмент не пропонує комплексного рішення, яке б поєднувало гнучкість ітеративної розробки з формалізмом MDE. Інструменти або пріоритезують гнучкість за рахунок формалізму (StarUML) [15], або забезпечують строгість ціною несумісності з сучасними практиками (Papyrus, Modelio) [103, 110], або створюють закриті екосистеми (Enterprise Architect, MagicDraw).

Відсутність готових промислових рішень обґрунтовує необхідність дослідження теоретичних підходів та методів, запропонованих у науковій літературі для вирішення проблеми консистентності. Аналіз цих методів, зокрема OCL [7, 94, 117, 146], теорії графів [29, 61, 85] та формальної верифікації [10. 38, 56, 46], є предметом наступного підрозділу.

1.5 Аналіз методів та підходів до перевірки консистентності моделей

Попередній аналіз встановив, що проблема консистентності є наслідком складності стандарту UML 2.x [5, 73, 139]. Цей виклик MDE поглиблюється конфліктом парадигм зберігання моделей: формалізму XMI [72, 155] та гнучкості

JSON [48, 127, 108]. Огляд ринку CASE-засобів емпірично довів, що жоден промисловий інструмент не вирішує цю дилему. Це змушує розробників обирати між гнучкими, але семантично обмеженими форматами, та формально строгими, але несумісними з сучасними інженерними практиками стандартами [58, 134].

Виявлена інструментальна прогалина вимагає переходу на теоретичний рівень аналізу. Мета цього підрозділу – критично проаналізувати наукові методи, формалізми та алгоритми, запропоновані в академічній літературі для вирішення проблеми консистентності [7, 112]. Завдання полягає у виявленні їхніх сильних сторін, фундаментальних обмежень та компромісів, а не в простому переліку. Такий аналіз дозволить зрозуміти, чому ці теоретичні досягнення не були інтегровані в промислові інструменти, та обґрунтувати існування не лише інструментальної, а й глибокої науково-практичної прогалини [145]. Виявлення цього теоретичного вакууму стане відправною точкою для розробки нового підходу, що є предметом даного дисертаційного дослідження [26].

Для систематизації методів перевірки узгодженості вводиться формальна таксономія, заснована на оглядових роботах у цій галузі [50, 79, 90]. Вона дозволяє порівнювати підходи за ортогональними критеріями для виявлення їхніх ключових характеристик та обмежень [94].

Перший вимір класифікації стосується стратегії перевірки, за якою виділяють три основні підходи [113]:

– *Моніторинг*. Стратегія полягає у валідації моделі відносно набору заздалегідь визначених формальних правил, обмежень чи інваріантів. Типовим представником є мова OCL [117, 146].

– *Аналіз*. Алгоритмічні методи, що, на відміну від моніторингу, не перевіряють задані правила, а аналізують структуру чи поведінку моделі для виявлення дефектів, таких як блокування (deadlocks), недосяжні стани або цикли залежностей. Прикладами є аналіз досяжності в мережах Петрі [25, 56] та пошук контрприкладів через SAT-вирішувачі [38].

– *Конструювання*. Генеративні підходи, де консистентність гарантується самим процесом побудови [122]. Одна частина моделі генерується з іншої за

формальними правилами трансформації, і за умови коректності цих правил результат є узгодженим за визначенням. Прикладом є потрійні графові граматики (Triple Graph Grammars, TGG) [29, 61, 85].

Другий вимір класифікації – час виконання валідації, який визначає придатність методу для інтеграції в сучасні ітеративні процеси розробки:

– *Повна***.** Традиційний підхід, що передбачає повну перевірку моделі за вимогою або в контрольні моменти (наприклад, перед комітом). Його висока обчислювальна вартість робить його непрактичним для частого застосування на великих моделях [60].

– *Інкрементальна***.** Оптимізований підхід, який після внесення локальної зміни перевіряє лише залежні елементи моделі, а не всю її цілком. Це на порядки підвищує ефективність, що є критично важливим для великих промислових моделей [34, 75].

– *Активна***.** Найбільш просунутий підхід, що надає розробнику миттєвий зворотний зв'язок у фоновому режимі безпосередньо під час редагування. Такий режим є ідеальним для практик Agile/DevOps, дозволяючи виправляти помилки на ранніх етапах [138].

Ця класифікація виявляє фундаментальну суперечність. Сучасні інженерні практики, як Agile та DevOps, вимагають швидкого зворотного зв'язку, надаючи пріоритет активним та інкрементальним режимам. Водночас парадигма MDE для забезпечення формальної строгості тяжіє до декларативних підходів моніторингу (OCL) та генеративних підходів конструювання (TGG). Однак саме ці формально строгі методи демонструють найгірші показники продуктивності та найменшу придатність для реалізації в активному чи інкрементальному режимі [85, 94]. Це створює «методологічну прогалину»: відсутність єдиної стратегії, яка б ефективно поєднувала формальну потужність з обчислювальною ефективністю, необхідною для сучасних робочих процесів [19, 81].

OCL, стандарт від OMG, є ключовим інструментом для додання формальної строгості UML-моделям [47]. Як текстова декларативна мова, вона дозволяє специфікувати інваріанти, передумови та постумови, які неможливо виразити графічно. Інтеграція з метамоделлю UML робить OCL центральним інструментом у

стратегії моніторингу консистентності. Ключовою перевагою OCL є його декларативність: мова описує, що має бути істинним, а не як це перевіряти, підвищуючи рівень абстракції до бізнес-логіки системи [146].

Проте, попри свій статус, а також методологічну цілісність, OCL має фундаментальні обмеження, що знижують його практичну цінність у складних промислових системах [58].

Перший недолік – це обмеження виразної потужності. Теоретично OCL не здатен виражати транзитивне замикання бінарного відношення. Це є фундаментальною проблемою, оскільки критично важливі архітектурні властивості, як ациклічність ієрархій успадкування чи компонентних залежностей, за своєю природою є транзитивними [94, 148]. Перевірка інваріантів на кшталт «клас не може успадковувати сам себе» вимагає аналізу всього ланцюга відношень. Таким чином, нездатність OCL формально описати такі властивості залишає найнебезпечніші архітектурні дефекти поза межами верифікації [14].

Друга проблема – це продуктивність та масштабованість. Навіть виразні обмеження є обчислювально дорогими для великих промислових моделей. Повна перевірка набору OCL-інваріантів може тривати від хвилин до годин, що робить регулярну валідацію непрактичною в ітеративних процесах розробки [18, 60]. Серйозність цієї проблеми підтверджують спеціалізовані бенчмарки, такі як Train Benchmark, що демонструють низьку продуктивність стандартних інтерпретаторів та стимулюють пошук інкрементальних рішень [81]. Додатковим бар'єром є складність написання оптимізованих OCL-правил, що спонукає до досліджень їх автоматичної генерації за допомогою великих мовних моделей.

Отже, OCL створює «парадокс формалізму». Введений для підвищення строгості моделей, він не може повноцінно реалізувати цю мету через власні недоліки. Модель з OCL-обмеженнями створює у розробників хибне відчуття безпеки: вона виглядає формально верифікованою, проте найважливіші архітектурні дефекти залишаються невиявленими через обмежену виразність, а локальні перевірки є занадто повільними для частого використання. Це обґрунтовує потребу в методах, що виходять за рамки стандартного OCL, наприклад, шляхом трансляції моделі у

більш потужні формалізми [88, 94].

Потрійні графові граматики (Triple Graph Grammars, TGG) – це декларативний, заснований на правилах підхід для підтримки узгодженості між двома моделями, що розглядаються як графи. В основі TGG лежить опис трійки графів: вихідного (source), цільового (target) та кореспондентського (correspondence), що пов'язує їхні елементи. Правила TGG синхронно модифікують усі три графи для збереження консистентності [61, 29]. Ключова перевага підходу – автоматична генерація правил для прямої (source → target) та зворотної (target → source) трансформації з єдиної декларативної специфікації [85, 64].

Здатність до двосторонньої синхронізації робить TGG теоретично привабливим рішенням для забезпечення консистентності між різними поданнями системи (наприклад, UML-модель та вихідний код, або структурна та поведінкова частини моделі) [63]. TGG реалізують стратегію «консистентність за конструюванням»: консистентність не перевіряється, а гарантується самим процесом трансформації [29].

Проте, попри теоретичну потужність, TGG мають фундаментальні недоліки, що обмежують їхнє практичне застосування в сучасних ітеративних процесах розробки [23].

*Обчислювальна складність***.** Пошук застосовних правил TGG у великих графах є NP-складною задачею, що робить трансформацію обчислювально дорогою [77]. Хоча існують спроби оптимізації, наприклад, через зведення до цілочисельного лінійного програмування [85, 150], фундаментальна складність залишається високою. Це робить TGG повільними для великих промислових моделей.

*Недетермінізм***.** Правила TGG за своєю природою можуть бути недетермінованими: для однієї ситуації може існувати кілька застосовних правил, що веде до різних результатів. Гарантування детермінізму, необхідного для надійної синхронізації, вимагає додаткових і складних для перевірки статичних умов [49].

Непридатність для швидкої валідації. Поєднання високої складності та недетермінізму робить класичні TGG непридатними для валідації в реальному часі. Вони призначені для пакетних офлайн-трансформацій, а не для миттєвого зворотного зв'язку в IDE. Існуючі спроби інкрементальної синхронізації на основі TGG є

складними в реалізації та не досягають продуктивності, необхідної для «живої» перевірки [23, 59, 85].

Отже, TGG є потужним теоретичним інструментом для формального опису узгодженості, але на практиці вони занадто повільні та складні для інтеграції у швидкі ітеративні процеси сучасної програмної інженерії.

Для подолання обмежень стандартних підходів запропоновано методи, що транслюють UML-моделі у потужніші формалізми для аналізу спеціалізованими інструментами. Такі підходи забезпечують глибокий семантичний аналіз та строгі гарантії коректності, але кожен має свої компроміси [90].

Трансляція в логіку та SAT/SMT-вирішувачі (Alloy). Цей підхід передбачає трансляцію UML-діаграм класів та OCL-обмежень у специфікацію Alloy, що базується на реляційній логіці першого порядку [38]. Аналізатор Alloy за допомогою SAT-вирішувачів проводить вичерпний пошук екземплярів моделі, які задовольняють або порушують задані обмеження [46].

Ключова перевага підходу – здатність генерувати конкретний контрприклад: мінімальний набір об'єктів та відношень, що демонструє помилку, дозволяючи розробнику швидко зрозуміти її причину. Проте головним недоліком є проблема «вибуху простору станів». Аналіз в Alloy є обмеженим: він ведеться в межах заданого користувачем «скоупу» (наприклад, не більше 5 екземплярів класу). Це означає, що Alloy може довести наявність помилки, знайшовши контрприклад, але не може гарантувати її відсутність для моделі необмеженого розміру, що є суттєвим компромісом для промислових систем.

Трансляція в моделі процесів (мережі Петрі). Цей клас методів фокусується на аналізі динамічної поведінки шляхом перетворення поведінкових діаграм UML (діяльності, станів) у формалізм мереж Петрі [10]. Мережі Петрі є потужним інструментом для аналізу паралельних, асинхронних та розподілених систем [25, 56]. Такий підхід дозволяє формально верифікувати ключові динамічні властивості. Аналіз графа досяжності мережі Петрі дає змогу перевірити досяжність певних станів, відсутність блокувань та життєздатність операцій, тобто можливість їх виконання з будь-якого стану.

Обмеженням підходу є його вузька спеціалізація на поведінці. Трансформація в мережі Петрі зазвичай абстрагується від структурних обмежень та обмежень на дані, визначених, наприклад, в OCL. Сам процес перетворення є нетривіальним і може призвести до втрати семантичної точності.

В результаті, методи формальної верифікації вимагають компромісу між глибиною та обсягом аналізу. Alloy надає глибокий, але обмежений структурний аналіз, тоді як мережі Петрі пропонують глибокий поведінковий аналіз, абстрагуючись від структури. Жоден з цих підходів не є цілісним та масштабованим рішенням для одночасної верифікації структурних і поведінкових аспектів великих промислових моделей [38, 46].

Алгоритмічні підходи є прямою відповіддю на низьку продуктивність повних перевірок та спрямовані на ефективну валідацію в ітеративних процесах [19, 81]. Їхній фундаментальний принцип – уникнення повної перевірки моделі після кожної зміни через локалізацію аналізу лише на потенційно порушених частинах. Для цього застосовуються такі техніки, як кешування, використання валідних екземплярів моделі як «сертифікатів» коректності та «зрізи» моделі для виділення мінімально необхідних для перевірки підмножин.

Центральну роль в інкрементальній валідації відіграють графи залежностей [11]. У цьому підході модель представляється як граф, де вершини – це елементи моделі (класи, атрибути), а ребра – залежності між ними (наприклад, виклик операції, тип атрибута) [43]. Коли розробник змінює елемент, алгоритм виконує обхід графа від зміненого вузла для визначення множини зачеплених елементів. Повторна перевірка правил узгодженості запускається лише для цієї, зазвичай невеликої, підмножини. Цей підхід є основою для створення обчислювально ефективних систем валідації великих моделей.

Кінцевою метою інкрементальних підходів є активна перевірка узгодженості [3]. Такі системи застосовують інкрементальні алгоритми у фоновому режимі для безперервного аналізу моделі, надаючи розробнику миттєвий зворотний зв'язок в IDE [81, 138]. Це дозволяє виправляти неузгодженості в момент їх виникнення, що є критично важливим для високих темпів розробки в практиках Agile та DevOps.

Слід підкреслити, що інкрементальні алгоритми є механізмом перевірки, а не специфікацією правил. Вони потребують формальної основи – набору правил узгодженості, що слугує критерієм коректності. Отже, ці алгоритми не вирішують проблему виразності OCL [148, 94], але пропонують ефективний шлях подолання його низької продуктивності.

1.6 Формалізація та таксономія помилок неузгодженості

Аналіз методів перевірки консистентності виявив їхні сильні сторони та обмеження. Проте для розробки досконалішого підходу необхідно перейти від оцінки методів до формалізації об'єкта їх застосування – типових помилок неузгодженості. Оскільки ефективність верифікації залежить від чіткого визначення цільових дефектів [24], метою даного підрозділу є саме їхня формалізація та класифікація [94, 145].

В основі формалізації лежить розмежування між синтаксичними та семантичними помилками [50, 90, 113]. Синтаксична консистентність – це відповідність моделі правилам її метамоделі (наприклад, UML), і її порушення зазвичай автоматично виявляються CASE-засобами. Натомість семантична консистентність стосується логічної несуперечливості змісту моделі [26]. Помилки цього типу виникають, коли синтаксично коректні подання суперечать одне одному за значенням [79]. Наприклад, бездоганні синтаксично діаграми класів та послідовності можуть разом описувати неможливу поведінку.

Предметом даного дослідження є саме семантичні помилки на межі між структурним та поведінковим поданнями, які є невидимими для стандартних інструментів [14, 94]. Необхідність таксономії таких помилок випливає з обмежень існуючих методів (наприклад, нездатність OCL виражати глобальні властивості), що унеможливлює використання єдиної всеохоплюючої мови верифікації. Натомість потрібен інженерний підхід: ідентифікація критичних патернів помилок для розробки цільових методів їх виявлення. Отже, створення таксономії є стратегічним кроком, що перетворює абстрактну «проблему консистентності» на конкретний набір

дефектів. Ця таксономія стане фундаментом і набором вимог для методів та інструментів, що розробляються далі [113].

На основі аналізу наукових джерел, присвячених проблемам узгодженості UML-моделей [112], та узагальнення типових дефектів, що виникають при паралельній еволюції структурних та поведінкових артефактів [42, 84], пропонується наступна таксономія семантичних помилок. Вона охоплює чотири критичні категорії, що найчастіше призводять до архітектурного дрейфу та прихованих відмов [17].

Невідповідність сигнатур операцій.

*Формальне визначення***.** Помилка невідповідності сигнатур виникає, коли кількість, типи або порядок параметрів у виклику операції, змодельованому в поведінковому поданні (наприклад, у повідомленні на діаграмі послідовності), не відповідають визначенню (сигнатурі) цієї операції у відповідному класі в структурному поданні. Це є порушенням контракту інтерфейсу на рівні моделі, що гарантовано призводить до помилок на етапі генерації коду або виконання [136, 135].

*Обґрунтування***.** Консистентність між операціями в діаграмах класів та повідомленнями в діаграмах взаємодії є одним з найбільш фундаментальних правил консистентності, що розглядається в численних систематичних оглядах. Дослідження Торре та ін. ідентифікували та валідували набір з 52 обов'язкових правил узгодженості, серед яких перевірка сигнатур посідає центральне місце. Порушення цієї відповідності є прямим свідченням розсинхронізації між проектом статичної структури та проектом динамічної взаємодії [79].

Приклад.

– Структурне подання**:** Клас PaymentService містить публічний метод для обробки платежу: processPayment(transactionId: String, amount: Money, currency: String).

– Поведінкове подання**:** Об'єкт :OrderController надсилає повідомлення processPayment(transactionId: String, amount: Money) до об'єкта :PaymentService.

– Аналіз помилки**:** У виклику відсутній третій обов'язковий параметр currency: String. При генерації коду з такої моделі [121] або під час виконання системи цей [20] виклик призведе до помилки компіляції (у статично типізованих мовах) або помилки

часу виконання через невідповідність кількості аргументів [136].

«Висячі» посилання та порушення посилальної цілісності.

Формальне визначення. Помилка типу «висяче» посилання виникає, коли елемент у поведінковому поданні (наприклад, Lifeline, State, Action) посилається на елемент у структурному поданні (наприклад, Class, Attribute, Operation), який був видалений, перейменований або його унікальний ідентифікатор було змінено. Це є порушенням посилальної цілісності на рівні моделі, що робить поведінкову діаграму семантично не валідною [114].

Обґрунтування. Проблема підтримки посилальної цілісності є класичною в управлінні даними та моделями. В контексті еволюції програмних моделей (model co-evolution), де рефакторинг, такий як перейменування чи видалення класів, є звичайною практикою, ризик виникнення таких помилок є надзвичайно високим [84]. Невиявлені «висячі» посилання перетворюють поведінкову модель на застарілий артефакт, що не відображає актуальну архітектуру системи і не може бути використаний для аналізу чи генерації коду.

Приклад.

– Структурне подання. У моделі існує клас ShoppingCart.

– Поведінкове подання. Діаграма послідовності для оформлення замовлення містить лінію життя :ShoppingCart, яка представляє екземпляр цього класу.

– Еволюція. В рамках рефакторингу архітектор приймає рішення перейменувати клас ShoppingCart на PurchaseBasket у структурному поданні для кращої відповідності термінології домену.

– Аналіз помилки. Поведінкове подання не було оновлено синхронно. Лінія життя :ShoppingCart тепер є «висячим» посиланням, оскільки клас з такою назвою більше не існує в структурній моделі. Будь-яка спроба проаналізувати цю діаграму або згенерувати з неї тестовий сценарій призведе до помилки.

Використання застарілих типів даних.

Формальне визначення. Помилка використання застарілих типів даних є специфічним випадком порушення посилальної цілісності, коли поведінкова діаграма оперує з даними (наприклад, параметрами повідомлень, змінними в діаграмі

діяльності, атрибутами об'єктів), тип яких було змінено, замінено або визнано застарілим (deprecated) у структурному поданні [44].

Обґрунтування. Ця помилка заслуговує на окремий розгляд через її поширеність під час рефакторингу, спрямованого на покращення моделі предметної області, наприклад, при заміні примітивних типів (як String чи double) на більш виразні класи-значення (Value Objects). Дослідження, що розглядають ко-еволюцію моделей та їх екземплярів [42], підкреслюють важливість підтримки узгодженості типів для збереження семантичної коректності системи [145].

Приклад.

– Структурне подання. Клас Product має атрибут price: double.

– Поведінкове подання. Діаграма діяльності, що моделює розрахунок знижки, містить дію, яка приймає та повертає змінну типу double для представлення ціни.

– Еволюція. Для уникнення проблем з точністю обчислень з плаваючою комою та для більш явного представлення грошових сум, тип атрибута price у класі Product змінюється на спеціалізований клас Money { amount: Decimal; currency: Currency; }.

– Аналіз помилки. Діаграма діяльності продовжує оперувати з примітивним типом double. Це створює неузгодженість типів, яка призведе до помилки типізації при генерації коду або до логічної помилки під час виконання, оскільки інформація про валюту буде втрачена [78].

Порушення інкапсуляції та послідовності взаємодії.

Формальне визначення. Помилка порушення інкапсуляції виникає, коли поведінкова діаграма моделює взаємодію, що порушує правила доступу (видимості), визначені у структурній моделі. Найбільш поширеним проявом є виклик приватного (private) або захищеного (protected) методу ззовні класу-власника або його нащадків [136].

Обґрунтування. Інкапсуляція є одним з фундаментальних принципів об'єктно-орієнтованого проектування, що забезпечує приховування внутрішньої реалізації та захист стану об'єкта. Порушення цього принципу на рівні моделі свідчить про глибоку архітектурну неузгодженість та неправильне розуміння контрактів між компонентами системи. Систематичні огляди правил консистентності часто

включають перевірки видимості, оскільки такі порушення вказують на потенційні дефекти дизайну [135].

Приклад.

– Структурне подання. Клас BankAccount має публічний метод deposit(amount: Money) та приватний метод - _updateBalance(newBalance: Money), який не повинен викликатися ззовні, оскільки він не виконує необхідних перевірок валідності.

– Поведінкове подання. Зовнішній об'єкт :FraudulentActor надсилає повідомлення _updateBalance(Money.MAX_VALUE) безпосередньо до об'єкта :BankAccount.

– Аналіз помилки. Поведінкова модель описує взаємодію, яка є неможливою в реалізації, оскільки порушує правила видимості, встановлені в структурній моделі. Це свідчить про помилку в проектуванні взаємодії, яка могла б призвести до серйозної вразливості, якби була реалізована в коді.

Наведена таксономія систематизує ключові семантичні помилки, що виникають на межі структурного та поведінкового подань [17].

Помилки з представленої таксономії є не просто недоліками моделювання, а належать до небезпечного класу прихованих відмов. Це дефекти, закладені на етапі проектування, що не виявляються стандартними перевірками (наприклад, компіляцією) і залишаються неактивними доти, доки специфічні умови виконання не спричинять несподіваний збій системи.

Накопичення таких неузгодженостей є проявом архітектурного технічного боргу [13, 82, 124, 132]. Кожна невиправлена розбіжність між структурним та поведінковим поданнями – це борг, взятий заради швидкості розробки. «Відсотки» за ним накопичуються у вигляді ускладненого супроводу, складнощів рефакторингу та зростання ризику збоїв. З часом реальна архітектура системи відхиляється від задокументованої, що є симптомом архітектурного дрейфу [6, 26].

Критичність проблеми підтверджується економічними принципами програмної інженерії. Дослідження, починаючи з класичних робіт Баррі Боема і до сучасних даних NASA та IBM, емпірично доводять: вартість виправлення дефекту зростає експоненційно протягом життєвого циклу [24, 52, 68]. Усунення дефекту на етапі

експлуатації може коштувати в 10–100 разів дорожче, ніж на етапі проектування. Помилки з даної таксономії є саме такими дефектами проектного рівня. Не виявлені на етапі моделювання, вони гарантовано проявляться пізніше – під час тестування або експлуатації, спричиняючи фінансові та репутаційні втрати [65].

Таким чином, розробка методів раннього виявлення цих помилок є не питанням якості моделей, а стратегічною інвестицією у зменшення загальної вартості володіння продуктом. Це переводить проблему консистентності з теоретичної площини в площину управління ризиками та економічної доцільності, підкреслюючи практичну значущість даного дисертаційного дослідження [52].

У даному підрозділі було формалізовано проблему семантичної неузгодженості між структурним та поведінковим поданнями UML. Запропонована таксономія (невідповідність сигнатур, «висячі» посилання, порушення інкапсуляції та ін.) перетворює абстрактну проблему на вимірюваний набір дефектів [113], фактично формуючи базис для алгоритмізації процесів їх автоматичного пошуку. Аналіз їхньої критичності як прихованих відмов, що ведуть до накопичення архітектурного боргу [13, 132], обґрунтовує економічну доцільність раннього виявлення [24, 52], оскільки вартість усунення таких помилок зростає нелінійно з кожним наступним етапом життєвого циклу. Хоча таксономія не є вичерпною, вона охоплює найпоширеніші критичні дефекти. Формалізувавши об'єкт дослідження, ми завершуємо аналіз стану проблеми, що дозволяє перейти до висновків розділу та постановки завдань дослідження.

1.7 Постановка задач дослідження

Проведений системний аналіз обґрунтовує необхідність розробки нового науково-методичного апарату для забезпечення консистентності UML-моделей. Формальна постановка задачі може бути сформульована наступним чином.

Нехай єдина концептуальна UML-модель M описує архітектуру програмної системи та складається з двох подань: структурного M_S (у форматі JSON) та поведінкового M_B (у форматі XMI). В результаті, модель визначається як

впорядкована пара: $M=(M_S,M_B)$. Логічна консистентність між поданнями регулюється скінченною множиною правил (інваріантів) консистентності R.

Основна проблема полягає в тому, що в умовах ітеративної розробки будь-яка операція редагування, застосована до одного з подань, може порушити консистентність усієї моделі. При цьому повна перевірка всіх правил R є обчислювально дорогою операцією, що ускладнює її інтеграцію в динамічні процеси розробки та створює потребу в механізмах синхронізації.

Задача дослідження полягає у створенні науково-методичного апарату, що забезпечує гарантовану консистентність UML-моделі з двома поданнями. Цей апарат повинен базуватися на обчислювально ефективному методі інкрементальної валідації, що дозволяє уникнути повної перевірки моделі, та включати алгоритми двосторонньої синхронізації для автоматичного поширення змін між структурним та поведінковим поданнями з метою підтримки їхньої узгодженості.

1.8 Висновки до розділу 1

У даному розділі висвітлено теоретичні засади проблеми забезпечення консистентності UML-моделей у сучасній модельно-орієнтованій інженерії. На основі аналізу предметної області розглянуто причини виникнення фундаментального протиріччя між гнучкістю, якої вимагають ітеративні процеси розробки, та формалізмом, необхідним для забезпечення достовірності архітектурних моделей. Здійснено огляд та пошук ефективних напрямів подолання означеної проблеми.

Виконано аналіз еволюції стандарту UML та існуючих форматів зберігання моделей (XMI, JSON та альтернатив), показано їхні переваги та недоліки в контексті сучасних інженерних практик. Проведено системний аналіз ринку CASE-засобів та наукових методів валідації, визначено їхні базові обмеження. Обґрунтовано вибір напрямку досліджень в області розробки комбінованого підходу до представлення та валідації UML-моделей.

Аналіз розглянутих джерел дозволяє стверджувати, що на сьогоднішній день

задача усунення конфлікту між гнучкістю та формалізмом при роботі з UML-моделями ще не набула комплексного розв'язку. Також існує брак алгоритмічних та інструментальних засобів, що підтримують ефективну валідацію та синхронізацію моделей, представлених у комбінованому форматі. Залишаються значні перспективи підвищення ефективності та достовірності модельно-орієнтованої розробки шляхом інтелектуального поєднання різних форматів та застосування обчислювально ефективних методів валідації.

За результатами аналітичного огляду літературних джерел зроблено наступні висновки:

1. Проведено систематизацію проблеми консистентності UML-моделей, підкреслено її глибоку семантичну природу та критичні наслідки її порушення (семантичний розрив, архітектурний дрейф, накопичення «технічного боргу»), що мають значний негативний економічний ефект. Для перетворення абстрактної проблеми на вимірюваний набір дефектів розроблено формальну таксономію типових семантичних помилок (невідповідність сигнатур, «висячі» посилання, порушення інкапсуляції та ін.), яка слугує фундаментом для розробки цільових методів валідації.

2. Виконано порівняльний аналіз домінуючих форматів зберігання UML-моделей (XMI, JSON), виявлено їхні фундаментальні обмеження (несумісність XMI з VCS, семантична обмеженість JSON). Обґрунтовано науково-практичну потребу в розробці комбінованого підходу, що поєднує гнучкість JSON для статичної структури та формальну строгість XMI для динамічної поведінки.

3. Проведено огляд сучасних CASE-засобів, встановлено відсутність інструментів з нативною архітектурною підтримкою комбінованої (мультиформатної) моделі. Показано, що існуючі рішення є архітектурно монолітними, змушуючи робити компромісний вибір між гнучкістю та формалізмом, що підтверджує існування значної науково-практичної прогалини на ринку.

4. Розглянуто поширені наукові методи верифікації консистентності, виявлено їхні обмеження: недостатня виразна потужність та низька продуктивність OCL, обчислювальна складність графо-трансформаційних підходів, вузька спеціалізація та проблеми масштабованості методів формальної верифікації (Alloy,

мережі Петрі). Це доводить відсутність єдиного методу, що ефективно поєднує строгість, виразність та обчислювальну ефективність, та обґрунтовує необхідність розробки високоефективних методів інкрементальної валідації.

На підставі наведеного встановлено, що вирішення протиріччя, висвітленого у вступі, може бути досягнуто шляхом розробки нового науково-методичного апарату, що поєднує переваги форматів JSON та XMI в рамках єдиної моделі та застосовує обчислювально ефективні методи для забезпечення її консистентності.

Сформульовано задачу дослідження: створити науково-методичний апарат, що забезпечує гарантовану консистентність UML-моделі з двома поданнями (структурним M_S у форматі JSON та поведінковим M_B у форматі XMI). Цей апарат повинен базуватися на обчислювально ефективному методі інкрементальної валідації, що дозволяє уникнути повної перевірки моделі після кожної зміни, та включати алгоритми двосторонньої синхронізації для автоматичного поширення змін між поданнями з метою підтримки їхньої узгодженості.

Вирішення встановленої задачі шляхом розробки комбінованого підходу ускладнюється наступними проблемами:

1. Відсутність формальної метамоделі, що описує UML з двома поданнями (JSON та XMI) та встановлює строге відношення відповідності між ними.
2. Брак обчислювально ефективних методів інкрементальної валідації, адаптованих для моделі з двома поданнями та здатних забезпечити швидкий зворотний зв'язок в ітеративних процесах.
3. Відсутність достовірних механізмів двосторонньої синхронізації, що автоматично поширюють зміни між поданнями для підтримки їхньої узгодженості.
4. Відсутність інструментальних засобів, що реалізують такий підхід та інтегруються в сучасні робочі процеси розробки (IDE, CI/CD).

Розв'язанню наведених проблем та створенню ефективного науково-методичного апарату та програмних засобів для забезпечення консистентності UML-моделей з двома поданнями присвячено наступні розділи даного дослідження.

2 ФОРМАЛЬНА МЕТАМОДЕЛЬ UML З ДВОМА ПОДАННЯМИ ДЛЯ ПІДТРИМКИ КОНСИСТЕНТНОСТІ

2.1 Формалізація передумов та еволюція концепції гібридного формату

Проведений у першому розділі системний аналіз проблеми забезпечення консистентності UML-моделей виявив фундаментальну та невирішену дихотомію, що лежить в основі сучасних інструментальних засобів та практик модельно-орієнтованої інженерії. Вона полягає у глибокому конфлікті між двома протилежними, але однаково важливими вимогами: гнучкістю та зручністю для розробника, яку пропонують легковагі текстові формати на кшталт JSON, та формальною строгістю і семантичною повнотою, яку гарантує стандартизований, але громіздкий формат XMI. Було доведено, що відсутність інструментальних засобів, здатних комплексно вирішити цей конфлікт, є однією з першопричин накопичення прихованих семантичних дефектів та, як наслідок, значного зростання вартості розробки та супроводу програмних систем [52].

Тому, констатувавши існування цієї науково-практичної прогалини, першим логічним кроком у пошуку її вирішення став прагматичний інженерний підхід. Замість того, щоб обирати одну з компромісних крайнощів, було висунуто гіпотезу, що оптимальне рішення може полягати у прямому синтезі їхніх сильних сторін. Цей пошук компромісу призвів до розробки початкової концепції, яка стала відправною точкою даного дисертаційного дослідження – так званого «гібридного формату». Ця ідея, що полягала у поєднанні JSON та XMI в рамках єдиного артефакту, була вперше сформульована та досліджена в ранніх публікаціях автора.

Даний підрозділ має на меті детально реконструювати та обґрунтувати еволюційний шлях дослідницької думки: від цієї початкової, прагматичної концепції до фінального, теоретично обґрунтованого апарату метамоделі з двома поданнями. Для цього буде проведено глибокий аналіз ідеології, архітектури та передбачуваних переваг гібридного формату. Після цього буде виконано не менш ретельний критичний розбір його фундаментальних обмежень та недоліків, які виявилися

непереборними і стали каталізатором для розробки більш досконалого, формального та достовірного рішення. Все це дозволить не просто констатувати перехід між двома концепціями, а й продемонструвати логічну неминучість цього переходу як результату послідовного наукового пошуку.

2.1.1 Розробка концепції гібридного формату

На початковому етапі дослідження концепція гібридного формату розглядалася як перспективне компромісне рішення, здатне подолати обмеження, притаманні як чистому XMI, так і чистому JSON. Ідеологія цього підходу ґрунтувалася на принципі розділення відповідальності, де вибір формату для конкретного елемента моделі визначався його природою та життєвим циклом змін.

Розвиток концепції гібридного підходу базується на низці завдань, виявлених під час аналізу наявних рішень. Одне з найважливіших – це збалансувати сильні сторони двох форматів, не скочуючись при цьому до надмірної складності. Бо при спробі зберігати поведінку UML (наприклад, кінцеві автомати або діаграми діяльності) необхідно забезпечувати сувору структуру і високу деталізацію, що в класичному JSON вирішити не завжди можливо. Однак формати на кшталт XMI надлишкові у випадках, коли модель невелика і часто змінюється в процесі прототипування [108]. Тож центральна ідея гібридного формату полягала в тому, щоб зберігати різні за своєю сутністю частини UML-моделі у форматах, що найкраще відповідають їхнім характеристикам. Тобто даний підхід передбачав певно модульності, за якої прості елементи (класи, атрибути, операції) задаються у легко зчитуваному JSON-форматі, а для опису особливо деталізованих об'єктів (складних відносин, поведінкових діаграм) допускають варіант вбудованих XMI-фрагментів.

В результаті, модель поділялася на дві складові в межах єдиного файлу:

− *Структурна частина (JSON):* Ця частина, що становила основу документа, описувала статичну архітектуру системи – класи, їхні атрибути, операції, прості асоціації та структуру пакетів. Вибір JSON на той момент для цієї ролі був обґрунтований його високою читабельністю для людини, компактністю та, що

найважливіше, відмінною сумісністю з сучасними системами контролю версій, такими як Git [123]. Оскільки саме статична структура моделі є найбільш динамічною та часто змінюваною частиною в процесі ітеративної розробки, використання JSON дозволяло генерувати чисті, зрозумілі звіти про відмінності та значно спрощувало процес злиття змін від кількох розробників.

− *Поведінкова частина (XMI):* Для опису складних динамічних аспектів системи, таких як діаграм станів, діяльності, послідовностей, передбачалося використання фрагментів у стандартному форматі XMI. Ці фрагменти мали зберігатися як текстові рядки всередині спеціальних полів JSON-об'єктів. Завдяки цьому дозволялось зберегти формальну строгість та семантичну повноту специфікації UML 2.x для тих частин моделі, де це є критично важливим, забезпечуючи потенційну сумісність з промисловими CASE-засобами, орієнтованими на XMI.

Підсумовуючи, ідеологія гібридного формату полягала у спробі отримати найкраще з обох світів: зберігається можливість експорту в інструменти, що підтримують XMI, підвищується наочність «базових» полів, а також залишається простір для розвитку складної моделі (через розширення XMI-частини). Це рішення виглядало як інженерний компроміс, що поєднує швидкість прототипування та зручність супроводу з формальною потужністю стандарту.

Для наочної демонстрації структури та принципів роботи гібридного формату розглянемо кілька прикладів, що ілюструють його застосування на різних етапах ускладнення моделі. Ці приклади показують, як модель може еволюціонувати від простого структурного опису до комплексного артефакту, що включає поведінкові аспекти.

На початковому етапі моделювання, коли основна увага приділяється статичній структурі, гібридний формат дозволяє описувати класи та їхні атрибути в чистому JSON, а для формального визначення складних відношень, таких як асоціації, використовувати вбудований XMI-фрагмент. Це забезпечує читабельність основної частини моделі, зберігаючи при цьому формалізм для зв'язків.

```
{
  "_type": "UMLClass",
  "name": "Example",
  "attributes":,
  "complexRelations":        "<xmi:Relation        xmi:type=\"uml:Association\"
source=\"class1\" target=\"class2\" />"
}
```

У цьому прикладі основна інформація про клас Example та його атрибут username представлена у JSON-форматі. Водночас, поле complexRelations містить рядок, що є валідним XMI-фрагментом, який формально описує асоціацію між двома іншими класами.

По мірі розвитку проекту виникає необхідність у моделюванні динамічної поведінки об'єктів. Гібридний формат передбачав для цього можливість вбудовування повноцінних поведінкових діаграм, таких як машина станів, безпосередньо в JSON-опис відповідного класу.

```
{
  "_type": "UMLClass",
  "name": "User",
  "attributes":,
  "operations":,
  "behaviorXMI": "<uml:StateMachine xmi:id=\"sm1\" name=\"UserStateMachine\"> \
                  <region xmi:type=\"uml:Region\" xmi:id=\"r1\"> \
                  <subvertex        xmi:type=\"uml:State\"        xmi:id=\"s1\"
name=\"Inactive\"/> \
                  <subvertex        xmi:type=\"uml:State\"        xmi:id=\"s2\"
name=\"Active\"/> \
                  <transition     xmi:type=\"uml:Transition\"     xmi:id=\"t1\"
name=\"Activate\" \
                  source=\"s1\" target=\"s2\"/> \
                  </region> \
                  </uml:StateMachine>"
}
```

Тут JSON-об'єкт, що описує клас User, містить поле behaviorXMI. Його значенням є рядок, що містить повний XMI-опис машини станів UserStateMachine з

двома станами (Inactive, Active) та переходом між ними (Activate). В цьому випадку дозволялось логічно пов'язати структуру класу з його поведінкою, зберігаючи при цьому формальну точність опису цієї поведінки.

Кінцевою метою концепції була тісна інтеграція з інтегрованими середовищами розробки та забезпечення двосторонньої синхронізації між моделлю та кодом. Для цього було запропоновано використовувати унікальні ідентифікатори, які б слугували мостом між елементами моделі (як у JSON, так і в XMI частинах) та їхньою реалізацією у вихідному коді.

```
{
  "id": "uml:Item_12345",
  "type": "Class",
  "name": "User",
  "attributes":,
  "relations":   "<xmi:Relation   xmi:type=\"uml:Association\"   source=\"User\"
target=\"Account\"/>"
}
```

В цьому випадку код на Java виглядав наступним чином:

```
public class User {              // UML: id=uml:Item_12345
    private String email;        // UML: id=uml:Item_12346
    //...
}
```

У цьому прикладі кожен елемент у JSON-частині моделі має унікальний id. Ці ж ідентифікатори використовуються у спеціальних коментарях у вихідному коді. Передбачалося, що спеціалізований плагін для IDE зможе використовувати ці ідентифікатори для встановлення відповідності: зміна імені класу User в коді призвела б до автоматичного оновлення поля name в JSON-об'єкті з id uml:Item_12345, і навпаки.

Для практичної реалізації роботи з гібридним форматом було розроблено концептуальну дворівневу архітектуру системи. Ця архітектура мала на меті забезпечити узгоджену обробку обох частин моделі та їхню інтеграцію із зовнішніми

інструментами. В основі системи лежала дворівнева модель, у якій перший рівень відповідає за роботу з JSON-описами базових UML-елементів, а другий – за зберігання розширених поведінкових конструкцій у форматі XMI.

Графічне представлення цієї архітектури наведено на рис. 2.1.

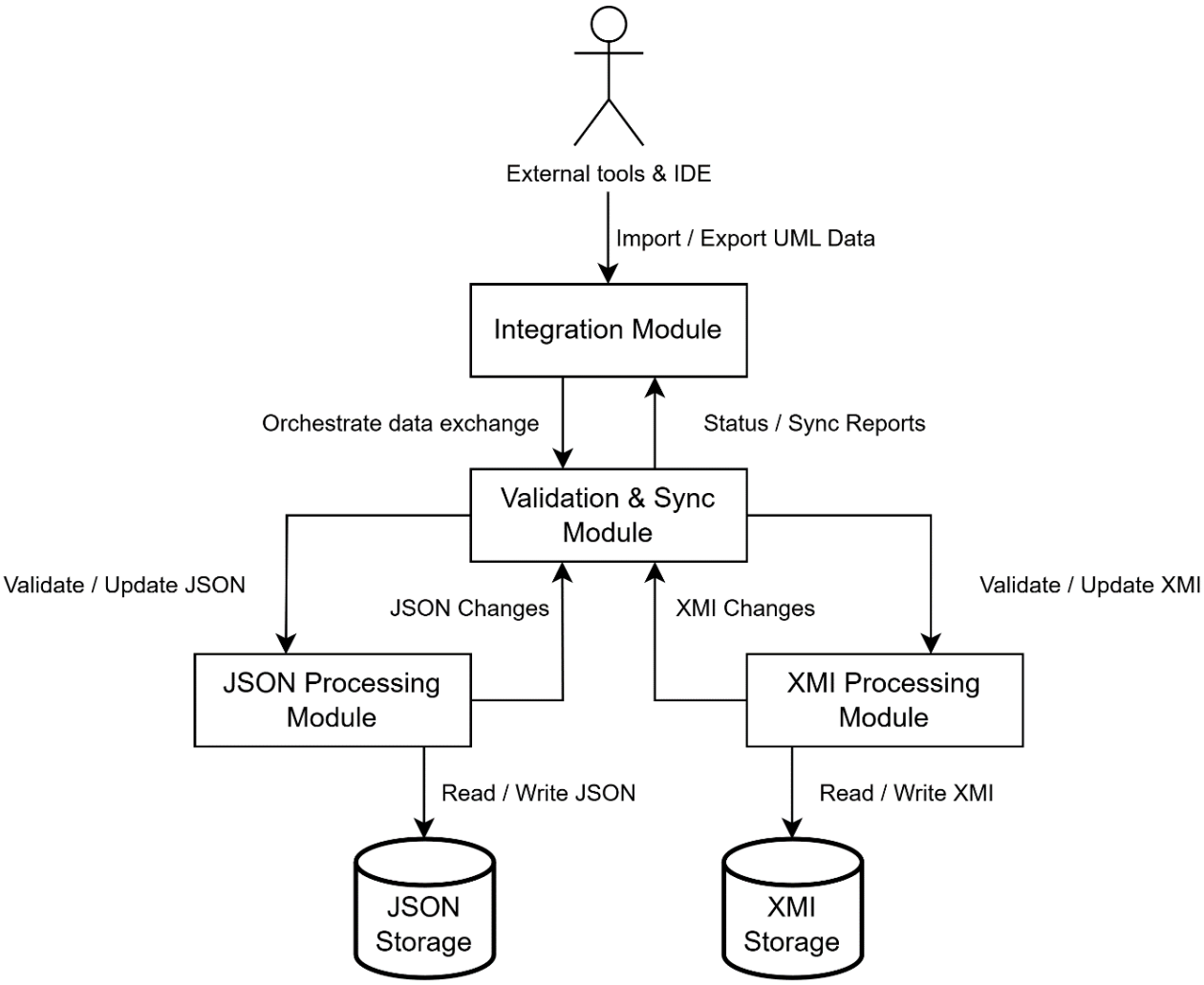

Рисунок 2.1 – Архітектурна модель системи для роботи з гібридним форматом

Ця архітектурна модель складалася з шести ключових функціональних блоків, кожен з яких виконував специфічну роль у життєвому циклі обробки гібридної моделі:

– *Сховище JSON-даних (JSON Storage).* Цей компонент відповідав за персистентне зберігання основної, структурної частини моделі. Він містив інформацію про класи, інтерфейси і прості відносини у вигляді JSON-файлів.

– *Сховище XMI-фрагментів (XMI Storage).* Паралельний компонент,

призначений для зберігання детально описаних діаграм станів, діяльності та інших складних елементів. Хоча в прикладах ці фрагменти показано як вбудовані рядки, архітектура передбачала можливість їхнього окремого зберігання для кращої керованості.

− *Модуль обробки JSON (JSON Processing Module).* Цей функціональний блок реалізовував логіку парсингу та серіалізації JSON-частини моделі. Він відповідав за читання JSON-файлів, побудову внутрішньопам'ятної об'єктної моделі структури та, навпаки, за збереження змін з цієї моделі назад у JSON-файл.

− *Модуль обробки XMI (XMI Processing Module).* Аналогічний за призначенням, але спеціалізований на роботі з XMI. Цей модуль мав відповідати за розбір і генерацію розгалужених поведінкових діаграм, що зберігалися у вигляді XMI-фрагментів.

− *Модуль валідації та синхронізації (Validation & Sync Module).* Це був концептуально найважливіший і, як виявилося згодом, найпроблемніший компонент архітектури, оскільки реалізація надійної двонаправленої синхронізації змін вимагала вирішення нетривіальних колізій. Його завданням було координувати взаємодію між JSON- і XMI-рівнями, стежачи за несуперечливістю посилань і відповідністю специфікації UML. Передбачалося, що цей модуль буде гарантувати цілісність усієї моделі. Наприклад, якщо в поведінковій діаграмі XMI є новий елемент, що відповідає методу класу, модуль перевіряє, чи є згадка про такий метод у JSON-частині. Саме існування цього модуля в початковій архітектурі свідчить про усвідомлення того, що консистентність не є автоматичною властивістю формату і вимагає спеціального механізму підтримки.

− *Модуль інтеграції із зовнішніми інструментами (Integration Module).* Цей блок слугував шлюзом для взаємодії з існуючою екосистемою CASE-засобів та IDE. Він мав за потреби давати змогу трансформувати гібридний формат у чистий XMI або в JSON-структури для подальшого використання в StarUML, EnterpriseArchitect, PlantUML і IDE.

Отже, запропонована архітектура являла собою комплексну систему, що мала на меті забезпечити повний цикл роботи з гібридною моделлю, від її створення та редагування до валідації та інтеграції.

2.1.2 Переваги та недоліки гібридного формату

На основі описаної ідеології та архітектури було сформульовано низку очікуваних переваг та типових сценаріїв використання гібридного формату, які мали б значно покращити процес модельно-орієнтованої розробки.

Головною перевагою вважалася можливість гнучкого та швидкого моделювання на ранніх етапах проекту. Розробники могли б починати роботу зі створення JSON-моделі, яка б описувала лише базову структуру класів. Таку модель легко редагувати вручну, аналізувати за допомогою простих скриптів та ефективно версіонувати в системах контролю версій. Тобто спочатку для швидких змін та прототипування задіяно легкий JSON, а з наростанням складності в модель додаються XMI-фрагменти з детальною поведінкою. Цей підхід ідеально узгоджувався з ітеративними методологіями розробки.

Водночас на відміну від підходів, що повністю покладаються на JSON або інші неформальні представлення, гібридний формат дозволяв не жертвувати формалізмом там, де він є необхідним. Для критично важливих аспектів бізнес-логіки, які потребували детального моделювання поведінки, розробники могли вбудовувати повноцінні, валідні XMI-фрагменти, що забезпечували повну відповідність специфікації UML.

Гібридний формат позиціонувався як міст між різними типами інструментів. Для гнучких, JSON-орієнтованих редакторів, таких як StarUML, основна частина моделі була б нативною. Для потужних, корпоративних CASE-засобів, як Enterprise Architect, модуль інтеграції мав би на льоту генерувати єдиний, монолітний XMI-файл, об'єднуючи дані з обох частин гібридної моделі. Разом це дозволяло б використовувати переваги кожного інструменту на відповідному етапі життєвого циклу.

Кінцевою метою була реалізація двосторонньої синхронізації між моделлю та кодом. Завдяки унікальним ідентифікаторам, плагін для IDE міг би відстежувати зміни як у коді, так і в моделі. Наприклад, якщо розробник перейменовує клас, плагін за унікальним ідентифікатором знаходить відповідний елемент у JSON/XMI-

структурі та оновлює його, забезпечуючи консистентність даних. Це мало б вирішити одну з головних проблем MDE – розсинхронізацію між проектними артефактами та їхньою реальною імплементацією [26].

Тому на етапі формулювання концепції гібридний формат виглядав як всеохоплююче та збалансоване рішення, що обіцяло поєднати гнучкість, формалізм та глибоку інтеграцію в єдиній, зручній для розробника екосистемі.

Але, незважаючи на початкову привабливість та формальну коректність, подальший, більш глибокий аналіз концепції гібридного формату виявив низку фундаментальних, непереборних вад, які були не поверхневими недоліками реалізації, а глибокими архітектурними прорахунками, що робили побудову достовірного, масштабованого та, що найголовніше, верифікованого рішення на його основі неможливим. Цей етап критичного переосмислення став поворотним моментом у дослідженні, який продемонстрував, що простий синтез форматів не є вирішенням проблеми, і спонукав до пошуку рішення на вищому, більш абстрактному рівні.

Формальна слабкість. Перший і головний недолік гібридного формату полягає в тому, що він не забезпечує справжньої, семантичної інтеграції між двома моделями даних. Просте вбудовування XMI-рядка в JSON-властивість є лише актом контейнеризації. JSON-структура виступає як пасивний контейнер для непрозорого, чужорідного блоку даних. З точки зору формальних моделей, між двома частинами артефакту відсутній будь-який визначений, машинооброблюваний зв'язок. JSON-парсер, обробляючи такий документ, бачить значення властивості behaviorXMI (що описує поведінку в форматі XMI) як звичайний рядок [48]. Він не має жодного уявлення про те, що цей рядок містить структуровані дані у форматі XMI, не кажучи вже про семантику UML. Для нього вміст цього рядка є атомарним та позбавленим внутрішньої структури. З іншого боку, якщо вилучити цей рядок і передати його XML-парсеру, той зможе обробити його як валідний XMI-фрагмент [155], але він робитиме це у повному вакуумі, не маючи жодного доступу до зовнішнього JSON-контексту, в якому цей фрагмент існував. Таким чином, гібридний формат створює два семантично ізольовані світи, які синтаксично поєднані, але логічно роз'єднані.

Провал валідації. Ця фундаментальна семантична ізоляція робить реалізацію достовірного механізму валідації, передбаченого в початковій архітектурі, практично неможливою. Розглянемо ключове завдання, яке мав би виконувати модуль валідації: перевірка посилальної цілісності між двома поданнями [79].

Уявімо типовий сценарій: елемент Lifeline у вбудованому XMI-фрагменті посилається на клас User, який, згідно з концепцією, має бути визначений у зовнішній JSON-структурі. Як валідатор може перевірити, що клас User дійсно існує в JSON-частині моделі? Стандартний валідатор (наприклад, на основі JSON Schema) бачить XMI-фрагмент як непрозорий рядок. Він не може зазирнути всередину цього рядка, знайти там посилання на User і перевірити його існування в іншому місці JSON-документа. Спеціалізований XMI-валідатор, отримавши на вхід рядок з XMI-фрагментом, може перевірити його внутрішню коректність. Однак він працює в ізольованому контексті і не має доступу до JSON-документа. Для нього посилання на клас User є зовнішнім і нерозв'язним.

В результаті виходить так, що не існує єдиного валідаційного контексту, в якому можна було б одночасно бачити та аналізувати структури обох форматів. Будь-яка спроба реалізувати таку перевірку зводилася б до написання складних, крихких та нестандартних парсерів, які б намагалися вручну зіставляти елементи з двох абсолютно різних моделей даних. Таке рішення було б вкрай неефективним і схильним до помилок.

Колапс трасування та цілісності. Неможливість достовірної валідації безпосередньо призводить до неможливості реалізації надійного трасування та автоматичної підтримки цілісності моделі під час її еволюції [84]. Розглянемо типову операцію рефакторингу: розробник перейменовує клас User на Customer у JSON-частині моделі.

Оскільки між JSON-структурою та вмістом XMI-рядків відсутній формальний, машинооброблюваний зв'язок, система не має жодного достовірного механізму для автоматичного пошуку та оновлення всіх входжень рядка User у десятках потенційно вбудованих XMI-фрагментів. Виконати таку заміну за допомогою простого пошуку та заміни тексту є вкрай небезпечним, оскільки це може випадково змінити інші

елементи, що містять такий самий підрядок.

Це неминуче призводить до масового виникнення «висячих посилань». Поведінкові діаграми, що зберігаються в XMI, миттєво стають застарілими та семантично некоректними, оскільки вони продовжують посилатися на неіснуючий клас User. З кожною такою операцією рефакторингу модель деградує, а її поведінкова частина перетворюється на марний, неактуальний артефакт, що лише вводить в оману розробників. Це повністю нівелює одну з головних цілей моделювання – підтримку актуальної та узгодженої проектної документації.

Підсумовуючи вищесказане, можна зробити висновок, що гібридний формат, попри свою привабливість для простих, статичних прикладів, є фундаментально крихким та немасштабованим рішенням. У будь-якому реальному проекті, де модель складається з сотень елементів, постійно змінюється та підтримується командою розробників, такий підхід призвів би до некерованого хаосу. Будь-яка зміна у структурній частині несе в собі високий ризик зламати поведінкову частину без будь-яких попереджень з боку інструментальних засобів. А ручне підтримання узгодженості між двома поданнями у великій моделі є надзвичайно трудомістким і схильним до помилок процесом [90], що повністю нівелює переваги у швидкості створення прототипів.

Провал концепції гібридного формату став ключовим уроком у дослідженні. Він продемонстрував, що проблема консистентності є не синтаксичною, а глибоко семантичною, і її неможливо вирішити шляхом маніпуляцій на рівні файлових форматів. Просте вкладення одного формату в інший не створює семантичного зв'язку, так само як розміщення двох книг різними мовами в одній коробці не робить їхній зміст узгодженим. Для вирішення виявлених фундаментальних проблем необхідно було перейти на вищий рівень абстракції.

Нова концепція, що виникла як пряма відповідь на недоліки гібридного підходу, отримала назву *метамодель з двома поданнями*. Цей підхід зберіг початкову ідею розділення відповідальності, але реалізував її на принципово іншому, теоретично обґрунтованому рівні. Замість того щоб розглядати JSON та XMI як частини одного файлу, нова концепція розглядає їх як два різні синтаксиси (конкретні

подання) для однієї, єдиної та логічно цілісної абстрактної моделі [21]. Тут можна виділити декілька ключових відмінностей. По-перше, замість одного файлу, що містить вбудовані рядки, нова архітектура передбачає два фізично окремі, але логічно пов'язані артефакти, що дозволяє використовувати для кожного з них стандартні, оптимізовані інструменти. По-друге, ненадійний механізм зв'язку через вбудовування рядка замінюється на строге математичне відношення відповідності. Воно формально визначає, які елементи з одного подання відповідають яким елементам з іншого. Саме це відношення стає тим семантичним мостом, якого бракувало гібридному формату і є центральним елементом метамоделі, що дозволяє формально описувати та перевіряти залежності між поданнями. Додатково варто виділити, що модуль валідації, який був неможливо реалізувати для гібридного формату, замінюється на систему формальних інваріантів, визначених для всієї метамоделі. Ці інваріанти, що можуть бути виражені декларативними мовами [47], діють на єдиній концептуальній моделі, що об'єднує обидва подання через відношення відповідності, що дозволяє виконувати строгу, автоматизовану та верифіковану перевірку глобальної консистентності, гарантуючи посилальну цілісність, відповідність сигнатур та інші критичні властивості моделі.

Тому, перехід до метамодельного підходу дозволив зберегти початкові переваги розділення, але при цьому забезпечити їхню достовірну взаємодію, а саме доопрацювання до теоретично стійкої архітектури став основою для всього подальшого науково-методичного апарату, розробленого в даній дисертаційній роботі.

2.2 Формалізація метамоделі

Для створення формально обґрунтованого та розширюваного рішення в основі дослідження лежить розробка не просто моделі, а саме метамоделі. Цей вибір є стратегічним рішенням, що переносить проблему з площини разового парсингу даних у площину інженерії мов. Замість створення інструменту з жорстко закодованими правилами узгодженості, розробка метамоделі означає визначення формальної мови,

для якої файли у форматах JSON та XMI є лише двома різними варіантами конкретного синтаксису. Такий підхід дозволяє визначити правила консистентності як невід'ємну частину самої мови, що відкриває шлях до автоматизованої верифікації, трансформації та генерації коду засобами MDE.

Фундаментальною основою для такого підходу є стандарт Meta-Object Facility (MOF), розроблений консорціумом OMG як архітектура метамоделювання для визначення мов, зокрема UML. MOF пропонує строгу чотирирівневу архітектуру, де кожен елемент на певному рівні є екземпляром елемента з вищого рівня:

− *Рівень M3 (мета-метамодель).* Верхній рівень, що визначає мову для побудови метамоделей. Сама специфікація MOF є M3-моделлю. Вона є рефлексивною, тобто може описувати саму себе, що усуває потребу в рівні M4.

− *Рівень M2 (метамодель).* Екземпляр M3-моделі. На цьому рівні визначаються мови моделювання. Класичним прикладом M2-моделі є метамодель UML, яка визначає такі поняття, як Class, Attribute, Association тощо. Саме на цьому рівні знаходиться запропонована в дисертації метамодель.

− *Рівень M1 (модель).* Екземпляр M2-моделі. Це рівень конкретних моделей, створених користувачами. Наприклад, діаграма класів для системи електронної комерції, що використовує поняття, визначені в метамоделі UML, є M1-моделлю.

− *Рівень M0 (об'єкти користувача, дані).* Екземпляр M1-моделі. Це рівень реальних об'єктів або даних системи під час її виконання. Наприклад, конкретний об'єкт customer:Customer є екземпляром класу Customer з M1-моделі.

Принцип суворого метамоделювання, закладений в основу MOF, гарантує, що кожна M1-модель точно відповідає правилам, визначеним у її M2-метамоделі. Саме тому розробка рішення на рівні M2 є єдиним шляхом для досягнення поставлених цілей. Лише метамодель надає необхідний рівень абстракції та формалізму, виступаючи як граматика або формальний контракт. Це дозволяє гарантувати однорідність та коректність цілої множини конкретних UML-моделей (рівень M1), забезпечуючи їх відповідність єдиним правилам, забезпечити можливість автоматизованої обробки**,** таких як валідації, трансформації моделей та генерації коду, оскільки інструменти MDE оперують саме на рівні M2-визначень, а також

інтегрувати різні формати подання (у даному випадку JSON та XMI) в єдиний, логічно цілісний формальний простір, де вони розглядаються не як окремі файли, а як різні серіалізації однієї й тієї ж концептуальної метамоделі [26].

Тому обравши метамодельний підхід, проблема забезпечення консистентності перетворюється на строгу інженерну дисципліну, засновану на міжнародних стандартах та формальних методах.

Архітектура запропонованої метамоделі базується на принципі розділення відповідальності. Єдина концептуальна модель фізично реалізується у вигляді двох окремих, але семантично пов'язаних подань, кожне з яких оптимізовано для вирішення специфічних завдань:

– *Структурне подання***.** Описує статичну архітектуру системи. Це можуть бути класи, інтерфейси, їхні атрибути, операції та відношення. Для його зберігання використовується формат JSON. Як вже було сказано, цей вибір обґрунтований його простотою, компактністю, ефективністю в керуванні пам’яттю та сумісністю з сучасними розподіленими системами контролю версій.

– *Поведінкове подання***.** Описує динамічні аспекти системи. Це можуть бути діаграми станів, послідовностей, діяльності, що моделюють взаємодію об'єктів, потоки керування та зміни станів. Для його зберігання використовується стандартний формат XMI. Вибір зумовлений його формальною строгістю, семантичною повнотою та стовідсотковою відповідністю специфікації UML 2.x. XMI гарантує сумісність з існуючими CASE-засобами та є основою для застосування формальних мов обмежень.

Такий підхід не є штучним технічним рішенням, а спирається на фундаментальні принципи архітектурного проектування, що підтверджується його прямою відповідністю класичній архітектурній моделі «4+1», запропонованій Філіпом Крухтеном, про яку йшла мова в першому розділі. Запропоноване в дисертації розділення моделі відображається на два ключові погляди моделі «4+1», де структурне подання реалізує логічне подання, а поведінкове подання – процесне подання. Завдяки цьому не просто виконується розділення статики та динаміки, а відбувається створення спеціалізованих, оптимізованих артефактів для різних ролей

у процесі розробки, дозволяючи кожному учаснику процесу використовувати інструменти та формати, що найкраще відповідають його задачам.

Визначення. UML-метамодель з двома поданнями – це формальне представлення метамоделі UML 2.5, яке складається з трійки згідно формулі

$$M = (M_S, M_B, \mu), \tag{2.1}$$

де M_S – структурне подання метамоделі, M_B – поведінкове подання метамоделі, а μ – відношення відповідності між елементами цих двох подань. Інтуїтивно, M_S визначає всі допустимі сутності та зв'язки статичної структури UML-моделі (класи, атрибути, асоціації, тощо), M_B – всі сутності для опису поведінки (активності, стани, послідовності повідомлень, тощо), а μ фіксує, як об'єкти структурного подання узгоджуються з об'єктами поведінкового подання. Метамодель побудовано за специфікацією UML 2.5 з урахуванням стандартних абстракцій (наприклад, клас Class, асоціація Association, стан State, перехід Transition, повідомлення Message тощо) та використанням поділу моделі на два подання

Кожен компонент має чітке математичне визначення, яке буде представлено нижче. Ця формалізація перетворює абстрактні ідеї про розділення моделі на точну, однозначну та машинооброблювану структуру, що є фундаментом для всіх подальших методів та алгоритмів, розроблених у дисертації.

Як вже було сказано, структурне подання M_S описує статичну архітектуру системи, тобто ті її аспекти, що залишаються незмінними під час виконання програми. Воно фіксує ключові будівельні блоки системи та відношення між ними, слугуючи основою для генерації коду та статичного аналізу. Формально, структурне подання визначається як пара

$$M_S = \langle T_S, R_S \rangle, \tag{2.2}$$

де T_S – скінченна множина допустимих типів елементів, а R_S – скінченна множина структурних відношень, що визначають зв'язки між цими елементами.

Множина типів T_S є ретельно відібраною підмножиною метакласів UML, яка включає лише ті елементи, що є необхідними та достатніми для опису статичної структури, яка зазвичай відображається у вихідному коді. Цей підхід дозволяє зменшити складність метамоделі та оптимізувати її для представлення у легкому форматі JSON.

Множина T_S визначається наступним чином:

T_S={ UMLClass, UMLInterface, UMLAttribute, UMLOperation, UMLParameter, UMLAssociation, UMLAssociationEnd, UMLGeneralization, UMLEnumeration, UMLPackage, UMLComponent, UMLObject, Primitive}

В свою чергу, множина відношень R_S визначає формальну семантику зв'язків між екземплярами типів з T_S. Ці відношення описують фундаментальні структурні залежності, такі як володіння, типізація та узагальнення.

Множина R_S включає наступні значення:

R_S={ hasAttribute, hasOperation, hasParameter, typedBy, generalizes, associates}

Кожне відношення має чітко визначену область визначення та область значень. Наприклад, відношення typedBy встановлює відповідність між атрибутом та його типом, який може бути як примітивним типом, так і іншим класом, визначеним у моделі. Формально це записується так:

− hasAttribute ⊆ UMLClass × UMLAttribute;

− hasOperation ⊆ UMLClass × UMLOperation;

− typedBy : UMLAttribute → (UMLClass ∪ Primitive);

− generalizes ⊆ UMLClass × UMLClass;

− associates ⊆ UMLAssociation × UMLClass.

Ця абстрактна математична формалізація слугує теоретичною основою та формальним контрактом для подальшої практичної реалізації структурного подання за допомогою JSON Schema, що буде детально розглянуто пізніше.

Поведінкове подання M_B описує динамічні аспекти системи, тобто її поведінку в часі та реакцію на зовнішні та внутрішні події. Воно моделює потоки керування, взаємодію об'єктів та зміни їхніх станів. За аналогією зі структурним поданням, поведінкове подання формалізується як пара:

$$M_B = \langle T_B, R_B \rangle, \tag{2.3}$$

де T_B – множина типів елементів, що відповідають метакласам UML для динамічних діаграм, а R_B – множина відношень, що визначають поведінкові зв'язки.

Множина типів T_B включає елементи, які за своєю природою є графовими та інтенсивно використовують перехресні посилання. Конструкції, такі як переходи між станами (Transition) або повідомлення між лініями життя (Message), погано піддаються опису в ієрархічних форматах і вимагають потужного механізму ідентифікаторів та посилань, який нативно підтримується у форматі XMI. Множина T_B визначається наступним чином:

T_B={Activity, Action, ControlFlow, StateMachine, State, Transition, Event, Interaction, Lifeline, Message, ExecutionSpecification, CombinedFragment}

Множина відношень R_B описує динамічні зв'язки причинно-наслідкового характеру, такі як виклик операцій, ініціювання переходів або вміст одних поведінкових елементів в інших. Множина R_B включає:

R_B={contains, triggers, represents, calls}

Наприклад, відношення calls пов'язує повідомлення, надіслане в діаграмі послідовності, з конкретною операцією, визначеною у структурному поданні, яку це повідомлення викликає. Формально: calls: Message → Operation. Це відношення є одним з ключових містків між поведінковим та структурним поданнями, консистентність якого гарантується за допомогою відношення відповідності μ.

Відношення відповідності μ є центральним механізмом запропонованої метамоделі, що забезпечує логічну зв'язність та семантичну цілісність між фізично розділеними структурним та поведінковим поданнями. Воно виступає як формальний контракт, що визначає правила взаємодії між двома світами: світом статичної структури та світом динамічної поведінки [79].

Формально, відношення μ визначається як підмножина декартового добутку множин екземплярів елементів структурного та поведінкового подань:

$$\mu \subseteq E_S \times E_B, \tag{2.4}$$

де E_S – множина всіх екземплярів елементів, типи яких належать до T_S, а E_B – множина всіх екземплярів елементів, типи яких належать до T_B. Кортеж $(s,b) \in \mu$ означає, що поведінковий елемент $b \in E_B$ семантично пов'язаний зі структурним елементом $s \in E_S$ і залежить від нього. Наприклад, якщо $l \in E_B$ – це лінія життя (Lifeline) в діаграмі послідовності, а $c \in E_S$ – це клас (Class), то кортеж $(c,l) \in \mu$ означає, що лінія життя l представляє екземпляр класу c. Аналогічно, кортеж $(o,m) \in \mu$, де o – операція, а m – повідомлення, означає, що повідомлення m моделює виклик операції o.

Для гарантування семантичної коректності та запобігання типовим помилкам неузгодженості, на відношення μ накладаються наступні фундаментальні властивості:

– *Частковість щодо поведінки*. Ця властивість є формальною гарантією того, що жоден поведінковий елемент, який семантично залежить від структури, не може існувати у вакуумі. Кожен такий елемент зобов'язаний мати посилання на структурний елемент, що запобігає появі «висячих посилань». Формально це можна описати наступним чином: $\forall b \in E_B, \exists s \in E_S : (s,b) \in \mu$. Це правило відображає фундаментальний принцип: поведінка є похідною від структури. Неможливо змоделювати виклик неіснуючої операції або взаємодію з неіснуючим класом.

– *Тотальність щодо структури*. Кожен структурний елемент, який позначений як hasBehavior та для якого можлива поведінкова деталізація, має хоча б один відповідник $b \in E_B$. Це обумовлено процесом розробки. Наприклад, прості класи для передачі даних або класи-переліки можуть не мати складної поведінки, що вимагає моделювання. Тому це дозволяє розробляти структуру системи, а потім поступово деталізувати поведінку лише для тих компонентів, де це необхідно, не порушуючи при цьому глобальної консистентності моделі.

– *Нефункціональність $E_S!\to!E_B$ та функціональність навпаки*. Один структурний елемент може мати кілька поведінкових відповідників, тоді як кожен поведінковий елемент має єдиний структурний якір. Ця властивість усуває неоднозначність, що, в свою чергу, гарантує, що, наприклад, повідомлення в діаграмі послідовності відповідає одній і тільки одній операції, що є критично важливим для однозначної інтерпретації моделі та подальшої генерації коду. Формально це

описується наступним чином: $\forall b \in E_B, \exists! s \in E_S : (s,b) \in \mu$.

– *Ідентифікаційна зв'язаність*. У кожній парі (s,b) ∈ μ один елемент безпосередньо посилається на id іншого, що забезпечує однозначне трасування між рівнями.

Завдяки сукупності цих властивостей забезпечується перетворення відношення μ з простого відображення на механізм забезпечення цілісності, який є формальною основою для всіх правил валідації, що будуть визначені далі.

2.3 Формалізація структурного подання у форматі JSON

Кожна діаграма UML (класи, компоненти, об'єкти, пакети) моделюється як часткове відображення цієї загальної структури M_S, 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.

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 [48].

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.

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 і містить загальні метадані (версію, ідентифікатор, тип моделі), а також вказівки на колекції сутностей.

```
{
  "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.

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. 2.1). 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). Усі визначені типи підтримують механізм посилань ($ref) для забезпечення модульності та повторного використання елементів.

Таблиця 2.1 – Узагальненя типів у JSON Schema / OAS

Тип	Призначення	Ключові поля	Особливості валідації
1	2	3	4
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

1	2	3	4
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 [67], 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 [71].

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. 2.2.

Нижче визначено множину позначень для діаграми класів:

– C – множина класів (UMLClass);

– A – множина атрибутів (UMLAttribute);

– 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 с безпосередньо успадковує клас p»;

– означення множини предків $Ancestors(c) = \{ p \in C | (c, p) \in Gen \}$.

Таблиця 2.2 – 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	$\forall a_1, a_2 \in A$: $class(a_1) = class(a_2) \wedge name(a_1) = name(a_2) \Rightarrow a_1 = a_2$
σ_5	Асоціації не є петлями	$\forall r \in R$: $end_1(r) \neq end_2(r)$
σ_6	Валідність типів атрибутів	$\forall a \in A$: $type(a) \in C \cup \{String, Integer, Boolean, Float\}$
σ_7	Унікальність асоціацій між парами	$\forall r_1, r_2 \in R$: $(end_1(r_1), end_2(r_1)) = (end_1(r_2), end_2(r_2)) \Rightarrow r_1 = r_2$

Також було приведено 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 було 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. 2.3.

Для формалізації моделі пакету було введено 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;

– $id : E \rightarrow ID$ – функція, що відображає елемент у його унікальний ідентифікатор;

– $ID \subseteq \Sigma+$;

– $name : E \rightarrow \Sigma+$ – 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 $p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_k$, то $(p_1, p_k) \in contains^+$. Це замикання дозволяє перевірити, чи не містить пакет сам себе – прямо або опосередковано [77].

Таблиця 2.3 – 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).

Нижче наведено 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 є 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.

Необхідно визначити 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. 2.4).

Для формалізації моделі компонентів введено 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 і визначається у вигляді всіх можливих послідовних застосувань відношення залежності (композиція на себе k разів, $k \geq 1$).

Таблиця 2.4 – 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).

Нижче наведено 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 введено 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. 2.5).

Таблиця 2.5 – 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)) ∈ allowed_types(a)

Нижче вказано JSON-приклад двох 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":39},
		"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. Крім того, завдяки цьому забезпечується наскрізна простежуваність змін між визначенням типів та їхніми екземплярами, що, в свою чергу, є необхідною передумовою для коректної роботи інкрементальних алгоритмів валідації у розподілених проектах

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. 2.6).

Таблиця 2.6 – Узагальнення валідаційних правил

№	Група правил	Формальна/схемна реалізація	Призначення
1	2	3	4
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 консистентність: кожне class, supplierId, superClassId тощо має збігатися 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

1	2	3	4
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

Отже, 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 консистентність зв'язків, цілісність міжоб'єктних посилань, унікальність імен у вкладених просторах) стандартної декларативної виразності недостатньо.

Для таких перевірок застосовується розширена валідація через зовнішні процедури 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.

Узагальнення моделі валідації структурного подання

Отже, в рамках цього підрозділу було розроблено та формалізовано структурне подання у форматі JSON для окремих ключових типів діаграм UML: діаграми класів, діаграми пакетів, діаграми компонентів та діаграми об'єктів. Кожна з цих формалізацій включала специфічний набір позначень та інваріантів, необхідних для забезпечення коректності саме цього типу діаграми [54].

Проте, наукова цінність роботи полягає не лише у вирішенні цих конкретних випадків, але й у узагальненні отриманих результатів. Аналіз розроблених інваріантів для окремих діаграм дозволив виявити фундаментальні категорії валідаційних правил, які є спільними для структурного подання будь-якої UML-моделі.

Наприклад, вимога ациклічності, що є критичною для коректності моделі, проявляється у кожному конкретному випадку. Для діаграми класів – це відсутність циклів узагальнення, для діаграми пакетів – як ациклічність вкладеності, а для діаграми компонентів – як ациклічність залежностей. Це дозволило абстрагуватися від конкретних відношень і сформулювати узагальнене правило R-7 «Ациклічність ієрархій», яке вирішується єдиним класом алгоритмів (обхід графа, DFS) незалежно від типу діаграми.

Аналогічно, вимоги до унікальності ідентифікаторів , обов'язковості полів та унікальності елементів у колекціях були узагальнені у відповідні правила (R-1, R-2, R-4).

Результатом цього аналізу стала розробка узагальненої моделі валідації, яка формалізована у табл. 2.6 і представляє не просто суму правил, а дворівневу валідаційну структуру, що складається з:

1. Декларативного рівня. Охоплює ~70% інваріантів, що стосуються синтаксису, структури, типів даних та локальної унікальності (правила R-1, R-2, R-3, R-4, R-6, R-9).

2. Процедурного рівня. Охоплює ~30% складних інваріантів, які потребують глобального контексту, аналізу транзитивних замикань або міжоб'єктних зв'язків (правила R-5, R-7, R-8). Для їх перевірки застосовується інтерпретація моделі як графу з подальшим обходом [77].

Таким чином, було не просто розроблено структури для окремих діаграм, але й сформульовано та узагальнено єдиний підхід до їх валідації.

Ця узагальнена модель доводить свою життєздатність та розширюваність. Якщо виникне необхідність додати підтримку нових типів діаграм (наприклад, діаграм станів або діаграм активностей), не буде потреби розробляти механізм валідації з самого початку. Нові сутності будуть підпорядковуватись вже існуючим загальним правилам. Наприклад, вони так само вимагатимуть унікальних ID (R-2) та валідних імен (R-6). Якщо ж у них з'являться власні ієрархічні чи транзитивні обмеження (наприклад, перевірка досяжності станів), то для їх реалізації вже існує

відпрацьований механізм процедурної валідації, що вимагатиме лише додавання нового модуля перевірки до CI/CD-конвеєра [125].

Отже, розроблена узагальнена модель валідації є трансформованою і може бути адаптована для різних умов та розширень UML-подання, що і становить один з елементів наукової новизни даної роботи.

2.4 Формалізація поведінкового подання у форматі XMI

У межах цього дослідження розглядається обмежена, але достатньо репрезентативна підмножина поведінкових діаграм UML: діаграми станів, діаграми діяльності та діаграми послідовностей [40]. 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: $b \vDash Consistent(\mu(b), M_S)$, 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 [139], 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. Така архітектура профілю надає можливість гнучко налаштовувати глибину трасування, охоплюючи як прямі посилання на ідентифікатори, так і складніші логічні залежності між різнорідними компонентами системи. Це дозволяє зберігати семантичний контекст навіть при перенесенні моделі між різними інструментальними середовищами. Ці прив’язки реалізуються через теговані значення, що забезпечують трасування елементів M_B до M_S без втрати сумісності (табл. 2.7), формуючи, таким чином, захищений метаданий шар, необхідний для роботи алгоритмів інкрементальної синхронізації.

Таблиця 2.7 – Стереотипи та теги UML-профілю для трасування між поведінковим та структурним поданнями

Стереотип	Базовий елемент UML	Теги (тип)	Призначення / приклад використання
1	2	3	4
«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»	Всі	contextInfo: String	Додатковий семантичний опис, який не може бути збережений у стандартних UML-атрибутах.

1	2	3	4
«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 [60].

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 [131] 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, так і для інкрементальної валідації локальних змін [10].

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

$$\forall s_m \in StateMachine, \exists! s \in s_m.substates : isInitial(s). \quad (2.5)$$

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.

$$\forall s \in States, \nexists t_1, t_2 \in s.outgoing, t_1 \neq t_2 : \quad (2.6)$$
$$t_2.trigger = t_2.trigger \wedge t_1.guard = t_2.guard.$$

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

$$\forall s \in States, \exists \pi = \langle t_1, \dots, t_k \rangle : t_1.source = s_0, t_k.target = s. \quad (2.7)$$

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.

$$\forall trig \in Trigger, \exists s \in M_S : \mu(trig) = s. \quad (2.8)$$

Діаграма діяльності у UML визначає поведінку системи як орієнтований граф [77], 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) [116]. 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 [16].

У рамках перевірки валідності такої моделі можна виділити такі 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:

$$\forall a \in A_N,\ isAction(a) \Rightarrow (\exists f \in F_L : tgt(f) = a \vee \exists f \in F_L : src(f) = a). \quad (2.9)$$

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 консистентні:

$$\forall f \in F_L, isControlFlow(f) \Rightarrow C\big(src(f)\big) \wedge C\big(tgt(f)\big) \quad (2.10)$$
$$isObjectFlow(f) \Rightarrow O\big(src(f), tgt(f)\big),$$

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.

$$\forall p \in InputPin, \exists s \in M_S : \mu(p) = s \wedge type(p) = type(s). \quad (2.11)$$

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:

$$\forall d \in A_N, isDecisionNode(d) \quad (2.12)$$
$$(|\{f \mid src(f) = d\}| \geq 2 \wedge \forall f\ (src(f) = d$$
$$\Rightarrow \neg isEmpty(guard(f)))).$$

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:

$$\exists fn \in A_N : isFinalNode(fn) \wedge Reachable(init, fn). \quad (2.13)$$

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»:

$$\forall l \in Lifeline, \exists s \in M_S : \mu(l) = s. \tag{2.14}$$

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:

$$\forall m \in Message, \exists c \in M_S : \mu\big(receiver(m)\big) = c \wedge \exists op \in Operations(c): name(op) = name(m) \wedge params(op) = params(m). \tag{2.15}$$

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:

$$\forall (m_1, m_2) \in Message^2: depends(m_2, m_1) \Rightarrow end(m_1) \prec start(m_2), \tag{2.16}$$

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 [79].

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:

$$\mu(eM_B) = eM_S, \tag{2.17}$$

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.

Узагальнення механізму валідації та трасування подань

Вище було розроблено набори формальних інваріантів, що гарантують синтаксичну та семантичну коректність для обмеженої, але репрезентативної

підмножини поведінкових діаграм UML: діаграм станів, діаграм діяльності та діаграм послідовностей. По суті, ці інваріанти являють собою конкретні рішення для специфічних елементів моделювання.

Аналіз розроблених правил (зокрема, інваріантів TriggerCorrespondence, InputMatchesType, LifelineCorrespondence та MessageConsistency) виявив наявність загальної проблеми: значна частина перевірок вимагає узгодження елементів поведінкового подання з елементами структурного подання, яке зберігається окремо у форматі JSON. Тому виникла необхідність у створенні універсального механізму, здатного забезпечити такий зв’язок для різних типів поведінкових елементів.

Для вирішення цієї задачі було розроблено узагальнений підхід, який складається з двох ключових компонентів:

1. Спеціалізований UML-профіль. Він надає загальну синтаксичну інфраструктуру для включення додаткової інформації про структурну прив’язку безпосередньо в XMI-подання поведінкової моделі.

2. Узагальнений механізм трасування. Цей механізм формалізує відношення між M_B та M_S і реалізується через теговані значення профілю, зокрема стереотип «JsonRef» та тег jsonId.

Ці компоненти дозволили сформулювати узагальнену модель відповідності, яка формалізує вимогу до всіх структурно-залежних елементів поведінкової моделі. Нехай $P \subseteq M_B$ – множина поведінкових елементів, для яких визначено зв’язок з M_S, тоді має виконуватись узагальнене правило:

$$\forall p \in P, \exists! \, s \in M_S : \mu(p) = s. \tag{2.18}$$

Це правило стверджує, що для кожного поведінкового елемента, який потребує узгодження, існує єдиний структурний відповідник в M_S. Таким чином, специфічні інваріанти, такі як InputMatchesType або LifelineCorrespondence, стають частковими випадками застосування цієї узагальненої моделі. Інваріанти, які посилаються на M_S, 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.

Наукова цінність такого узагальнення полягає в тому, що розроблений підхід не обмежується лише трьома розглянутими типами діаграм. Він демонструє, що запропоновані рішення можна трансформувати та адаптувати для інших випадків. Наприклад, якщо у майбутньому виникне потреба додати підтримку інших типів поведінкових діаграм (зокрема діаграм синхронізацій або огляду взаємодій, які на даному етапі не розглядалися), не буде потреби розробляти механізм валідації та трасування «з нуля». Натомість можна буде застосувати існуючу узагальнену модель: розширити профіль на нові елементи UML та сформулювати для них нові специфічні інваріанти, які будуть спиратися на вже існуючий загальний механізм відображення μ та валідації у контексті M_S.

2.5 Висновки до розділу 2

Розроблено формальну метамодель UML з двома поданнями, визначену як трійку $M=(M_S, M_B, \mu)$. Цей підхід, що ґрунтується на стандарті MOF, переносить вирішення проблеми з рівня маніпуляції файлами на рівень інженерії мов. Такий підхід створює строге теоретичне підґрунтя, що розглядає JSON та XMI як різні синтаксиси для єдиної концептуальної моделі, та є основою для подальших методів валідації та трансформації.

В рамках запропонованої метамоделі було формально визначено структурне подання M_S для опису статичної архітектури (класи, атрибути, відношення) та поведінкове подання M_B для динамічних аспектів системи (діаграми станів, послідовностей, діяльності). Доведено, що такий поділ є архітектурно обґрунтованим, оскільки він відповідає ключовим поглядам класичної моделі «4+1» Філіпа Крухтена, де M_S реалізує логічне подання, а M_B – процесне.

Центральним елементом розробленої метамоделі є відношення відповідності μ, що формально визначається як підмножина декартового добутку елементів обох подань. Цей механізм слугує формальним «семантичним мостом», що забезпечує логічну зв'язність, посилальну цілісність та можливість однозначного трасування між

структурними та поведінковими елементами – ключовими властивостями, яких бракувало початковій концепції гібридного формату.

Для практичної реалізації структурного подання M_s розроблено уніфіковану JSON-структуру для ключових діаграм (класів, пакетів, компонентів). Коректність цієї структури гарантується набором формальних інваріантів, реалізованих через специфікацію на основі JSON Schema. Це дозволяє автоматизувати перевірку синтаксичної та структурної цілісності моделі, що є першим кроком до забезпечення її глобальної консистентності.

Для інтеграції поведінкового подання M_B (у форматі XMI) з відокремленою структурною моделлю було розроблено спеціалізований UML-профіль. Цей профіль вводить набір стереотипів та тегованих значень, які використовуються для реалізації відношення трасування μ. Такий підхід дозволяє зберігати зв'язки між поданнями, не порушуючи при цьому сумісності зі стандартом XMI та існуючими CASE-засобами.

Проведений аналіз та формалізація окремих подань дозволили сформулювати узагальнені підходи до їх валідації та трасування. Для структурного подання розроблено узагальнену дворівневу модель валідації (декларативну та процедурну), яка може бути розширена для підтримки нових типів діаграм шляхом додавання специфічних інваріантів до вже існуючої структури перевірки. Для поведінкового подання створено узагальнений механізм трасування на основі UML-профілю, що дозволяє підключати нові типи поведінкових діаграм до структурного подання, спираючись на єдиний підхід до відображення за допомогою μ. Ці узагальнені моделі доводять, що розроблений апарат є не лише коректним для розглянутих випадків, але й гнучким та масштабованим для подальшого розвитку.

Матеріали розділу опубліковані в роботах [106, 157, 159, 160, 163, 164, 166, 167].

3 МЕТОДИ ЗАБЕЗПЕЧЕННЯ КОНСИСТЕНТНОСТІ UML-МОДЕЛІ

У попередньому розділі був розроблений теоретичний апарат, який закладає надійний фундамент для управління консистентністю UML-моделей за рахунок статичного, математично строгого опису того, яким критеріям повинна відповідати узгоджена модель. Але у реальних умовах ітеративної розробки модель є динамічним артефактом, що безперервно еволюціонує. Практика повної перевірки всієї моделі після кожної зміни є обчислювально затратною та непрактичною, особливо в інтерактивних середовищах та конвеєрах безперервної інтеграції. Така неефективність пакетної валідації призводить до поступового накопичення неузгодженостей, які переростають у значний архітектурний технічний борг [93, 126]. Це зумовлює об’єктивну необхідність трансформації статичних теоретичних принципів у дієві інструменти, здатні працювати в режимі реального часу без суттєвих затримок.

Завданням даного розділу є розробка цілісного науково-методичного апарату, що забезпечує динамічну підтримку консистентності моделі [95]. В основу цього апарату покладено принцип мінімізації обчислювальних витрат при збереженні гарантій коректності, що реалізується через взаємодію спеціалізованих алгоритмів. Для досягнення цієї мети було створено три методи:

− метод автоматизованого забезпечення консистентності, що функціонує на основі формально визначеного відношення відповідності між структурними та поведінковими елементами та базується на застосуванні OCL-інваріантів для верифікації локальної та глобальної узгодженості моделей при інкрементальному редагуванні;

− метод інкрементальної валідації, що локалізує перевірку лише на тих елементах моделі, цілісність яких могла бути порушена останньою зміною;

− метод двосторонньої синхронізації, механізму, що не просто виявляє помилки, а й автоматично поширює зміни між структурним та поведінковим поданнями для збереження їхньої узгодженості в реальному часі;

3.1 Метод автоматизованого забезпечення консистентності

3.1.1 Система формальних обмежень

Для гарантування семантичної коректності та узгодженості будь-якої конкретної UML-моделі, створеної на її основі, недостатньо виконати лише математичний опис метамоделі. Модель може бути синтаксично правильною, тобто відповідати структурним правилам метамоделі, але при цьому містити глибокі логічні суперечності, такі як виклик неіснуючої операції або невідповідність сигнатур, що є першопричиною архітектурного дрейфу та накопичення технічного боргу [74]. Більш детальна класифікація суперечностей була наведена в підрозділі 1.6.2.

Тому для усунення цієї прогалини необхідно ввести систему формальних обмежень (інваріантів), які повинні виконуватися для будь-якого екземпляра моделі. Ці інваріанти описують правила, за якими статична структура метамоделі може коректно функціонувати, що перетворює перевірку консистентності з неформальної, схильної до помилок практики на строгу, верифіковану інженерну дисципліну.

Для специфікації формальних обмежень було обрано мову OCL. Цей вибір є стратегічним, оскільки OCL є невід'ємною частиною екосистеми UML та стандартом саме для вирішення подібних задач [47]. Серед ключових переваг OCL, що зумовили його застосування в даній роботі, є:

− *тісна інтеграція з UML***.** OCL розроблено для роботи з метамоделлю UML, що дозволяє природно та інтуїтивно переміщатися по елементах моделі та формулювати запити до їхніх властивостей та відношень;

− *декларативна природа***.** OCL є декларативною мовою, що дозволяє специфікувати *що* має бути істинним у моделі, а не *як* це перевіряти. Це підвищує рівень абстракції, наближаючи обмеження до бізнес-логіки системи та звільняючи їх від деталей реалізації алгоритмів перевірки;

− *формальність та однозначність***.** OCL надає формальний, текстовий синтаксис, що усуває неоднозначність, притаманну обмеженням, описаним природною мовою, але при цьому залишається значно більш читабельним та доступним для інженерів, ніж складні математичні формалізми [117].

На основі цього інструментарію було розроблено систему ключових інваріантів, що формалізують вимоги до узгодженості між структурним та поведінковим поданнями, які були вперше представлені в роботах автора і забезпечують семантичну цілісність запропонованої метамоделі.

Інваріант повноти відповідностей (BehaviorExists).

Цей інваріант гарантує, що кожна операція у структурному поданні, яка позначена як така, що має складну поведінку (за допомогою булевого атрибута hasBehavior = true), має відповідну реалізацію в поведінковому поданні. Це правило запобігає появі таких операцій, які задекларовані в архітектурі, але їхня логіка ніде не визначена, що є поширеним архітектурним дефектом.

Формальний опис на OCL виглядає наступним чином:

```
context Class
inv BehaviorExists:
  self.operations->select(op | op.hasBehavior)->forAll(op |
    Interaction.allInstances()->exists(i | i.messages->exists(m | m.operation = op)) or
    StateMachine.allInstances()->exists(sm | sm.transitions->exists(t | t.trigger.operation = op)) or
    Activity.allInstances()->exists(act | act.actions->exists(a | a.operation = op))
  )
```

Інваріант застосовується в контексті кожного класу (context Class). Він вибирає всі операції (operations), для яких встановлено прапорець hasBehavior (->select(op | op.hasBehavior)), і для кожної з них (->forAll(op |...)), перевіряє, чи існує (->exists(...)) відповідний їй елемент хоча б в одній з поведінкових діаграм: або повідомлення (Message) в діаграмі взаємодії (Interaction), або тригер переходу (Transition) в машині станів (StateMachine), або дія (Action) в діаграмі діяльності (Activity).

Важливо відзначити, що застосування атрибута hasBehavior є не формальною слабкістю, а прагматичним інженерним рішенням. Догматичний підхід вимагав би, щоб *усі* операції мали поведінковий відповідник. Однак у реальних системах існує велика кількість простих операцій, моделювання поведінки яких є надлишковим і

лише ускладнює модель. Введення прапорця hasBehavior дозволяє розробнику свідомо застосовувати строгу формальну верифікацію лише там, де вона є доцільною – для ключової бізнес-логіки.

Інваріант відсутності «висячих посилань» (AnchorExists).

Це фундаментальне правило цілісності, яке вимагає, щоб кожен поведінковий елемент мав відповідний структурний якір. Інваріант запобігає появі «висячих посилань», коли, наприклад, лінія життя в діаграмі послідовності посилається на клас, який був видалений або перейменований у структурному поданні.

Формальний опис на OCL виглядає наступним чином:

```
context BehaviorElement
inv AnchorExists:
  not self.anchor.oclIsUndefined() and
  StructuralElement.allInstances()->includes(self.anchor)
```

Інваріант застосовується до узагальненого типу BehaviorElement, що представляє будь-який елемент поведінкового подання. Він перевіряє дві умови: по-перше, що посилання на якір (self.anchor) взагалі визначене (not self.anchor.oclIsUndefined()), і по-друге, що об'єкт, на який воно вказує, дійсно існує у множині всіх екземплярів структурних елементів (StructuralElement.allInstances()->includes(self.anchor)).

Інваріант єдиності якоря (SingleAnchor).

Цей інваріант посилює попереднє правило, вимагаючи, щоб кожен поведінковий елемент мав *лише один* структурний якір. Це усуває неоднозначність, гарантуючи, що, наприклад, повідомлення в діаграмі послідовності відповідає одній і тільки одній операції, що є критично важливим для однозначної інтерпретації моделі та подальшої генерації коду.

Формальний опис на OCL виглядає наступним чином:

```
context BehaviorElement
inv SingleAnchor:
  StructuralElement.allInstances()->select(s | s.behavioralCounterpart = self)-
```

```
>size() = 1
```

Інваріант фільтрує (->select(...)) множину всіх структурних елементів, залишаючи лише ті, які посилаються на даний поведінковий елемент (self), і перевіряє, що розмір (->size()) отриманої колекції дорівнює одиниці.

Інваріант відповідності сигнатур (SignatureMatches).

Це одне з найважливіших правил семантичної узгодженості, яке гарантує, що сигнатура виклику операції в поведінковому поданні (наприклад, у повідомленні діаграми послідовності) точно відповідає її визначенню у структурному поданні (у класі). Перевіряється як кількість, так і типи параметрів, що запобігає типовим помилкам часу виконання.

Формальний опис на OCL виглядає наступним чином:

```
context Message
inv SignatureMatches:
  self.opRef.oclIsUndefined() or (
    self.arguments->size() = self.opRef.parameters->size() and
    Sequence{1..self.arguments->size()}->forAll(i |
      self.arguments->at(i).type.conformsTo(self.opRef.parameters->at(i).type)
    )
  )
```

Інваріант застосовується в контексті повідомлення (Message). Він ігнорується, якщо повідомлення не посилається на операцію (self.opRef.oclIsUndefined() or...). В іншому випадку, він перевіряє, що кількість аргументів повідомлення (self.arguments->size()) дорівнює кількості параметрів операції (self.opRef.parameters->size()). Після цього для кожної пари аргумент-параметр (->forAll(i |...)), він перевіряє, що тип аргумента відповідає (conformsTo) типу параметра.

Сукупність цих інваріантів формує формальний контракт, що гарантує консистентність між двома поданнями моделі на семантичному рівні.

Хоча OCL є потужною мовою для специфікації, її стандартні інструменти та сама мова мають фундаментальні обмеження, які ускладнюють перевірку певних класів властивостей [148, 94]. Це було детально проаналізовано в підрозділі 1.5.3.

Ключовим теоретичним недоліком є нездатність виражати транзитивне замикання бінарного відношення [14]. Це не є абстрактною академічною проблемою; це означає, що OCL не може нативно виразити такі критичні архітектурні інваріанти, як «клас не може успадковувати сам себе» (ациклічність ієрархії успадкування) або «компонент не може мати циклічної залежності від самого себе».

Для подолання цих обмежень та забезпечення більш глибокого семантичного аналізу, відповідно до висновків, зроблених у підрозділі 1.5.5, пропонується підхід, що полягає у трансформації моделі та її OCL-обмежень у специфікацію на мові Alloy [38].

Alloy – це декларативна мова, заснована на реляційній логіці, яка є більш виразною для певних структурних властивостей. Її аналізатор, Alloy Analyzer, не інтерпретує обмеження, а транслює всю специфікацію моделі у велику булеву формулу і використовує потужні SAT-вирішувачі для вичерпного пошуку екземплярів, що задовольняють або порушують задані обмеження [46].

Це надає дві фундаментальні переваги.

Перша – строга формальна верифікація. Аналіз на основі SAT-вирішувачів забезпечує математично строгу гарантію коректності моделі, хоча й в межах заданого користувачем невеликої області. Цей підхід спирається на гіпотезу малої області, яка стверджує, що більшість архітектурних дефектів можуть бути виявлені при аналізі невеликих екземплярів системи, що робить обмежену верифікацію високоефективною на практиці.

Друга – автоматична генерація контрприкладів. Це найбільш потужна практична перевага Alloy. У разі порушення інваріанта, модельний аналізатор не просто повідомляє про помилку, а генерує *контрприклад* – конкретний, мінімальний набір об'єктів та відношень, що наочно демонструє проблему. Для розробника це означає перехід від абстрактного повідомлення про помилку до аналізу конкретного сценарію відмови, завдяки чому процес налагодження моделей перетворюється з дедуктивного пошуку причини на прямий аналіз конкретного випадку, що значно скорочує час та зусилля на виправлення складних семантичних дефектів.

В результаті даний двоступеневий процес створює формальний верифікаційний

конвеєр: зручні для інженера OCL-обмеження транслюються у більш потужний формалізм Alloy для глибокого, автоматизованого аналізу.

Розглянемо, як інваріант SingleAnchor, визначений мовою OCL, транслюється у специфікацію Alloy.

OCL-запис:

```
Object Constraint Language
context BehaviorElement
inv SingleAnchor:
  StructuralElement.allInstances()->select(s | s.behavioralCounterpart = self)->size() = 1
```

Трансформація в Alloy:

```
-- Оголошення базових типів (сигнатур)
sig StructuralElement {}
sig BehaviorElement {
  -- Кожен поведінковий елемент має відношення 'anchor',
  -- яке пов'язує його з РІВНО ОДНИМ структурним елементом.
  anchor: one StructuralElement
}

-- Твердження (assertion) для перевірки інваріанта
assert SingleAnchorAssertion {
  all b: BehaviorElement | one s: StructuralElement | b.anchor = s
}

-- Команда для аналізатора: перевірити твердження
-- для моделей, що містять до 5 екземплярів кожного типу.
check SingleAnchorAssertion for 5
```

У цьому прикладі метакласи UML StructuralElement та BehaviorElement відображаються на сигнатури (sig) Alloy. Відношення між ними (anchor) моделюється як поле сигнатури. Ключовим моментом є те, що складна конструкція OCL ->select(...)->size() = 1 прямо виражається в Alloy за допомогою квантора кратності one. Це демонструє природність відображення між двома формалізмами та потужність

реляційної логіки Alloy для опису структурних обмежень.

Замість повної компіляції всієї підмножини OCL, застосовано підхід трансляції на основі шаблонів, де виконується розпізнавання критичних патернів проектування та обмежень у вихідній моделі, які потім відображаються у відповідних конструкціях Alloy.

Формалізація правил трансляції наведена у табл. 3.1.

Таблиця 3.1 – Формалізація правил трансляції

Категорія	Конструкція UML/OCL	Конструкція Alloy	Семантичний опис
Структура	Class User	sig User { ... }	Клас UML транслюється у атомарну сигнатуру, що позначає множину об'єктів.
Атрибути	User.name: String	field: set String	Атрибути та асоціації стають полями-відношеннями всередині сигнатури.
Інваріанти	context X inv: ...	`fact { all self: X	... }`
Кардинальність (1)	self.y->size() = 1	one this.y	Квантор one гарантує наявність рівно одного зв'язаного елемента.
Кардинальність (0..1)	self.y->size() <= 1	lone this.y	Квантор lone (less or equal one) дозволяє відсутність зв'язку або один зв'язок.
Існування	`coll->exists(e	P(e))`	`some e: coll
Навігація	self.a.b	this.a.b (Join)	"Крапкова" нотація OCL прямо відображається в операцію реляційного з'єднання (join) в Alloy.
Унікальність	allInstances->isUnique(id)	no disj a,b:X | a.id=b.id	Перевірка унікальності транслюється як заборона існування двох різних (disj) об'єктів з однаковим ID.
Спадкування	Class B extends A	sig B extends A { }	Alloy підтримує ієрархію сигнатур, що відповідає успадкуванню в UML.

Особливої уваги заслуговує трансляція навігаційних виразів та операцій замикання, які є складними для реалізації у традиційних SQL-подібних валідаторах. Вираз OCL self.department.manager.name семантично означає послідовний перехід по графу об'єктів. В Alloy це реалізується через оператор з'єднання ., який природним чином обробляє множини. Якщо department повертає множину об'єктів, то вираз department.manager автоматично поверне об'єднання менеджерів усіх цих департаментів. Тому це усуває необхідність у складних циклах forAll/collect, характерних для процедурних мов.

Однією з найбільш потужних можливостей Alloy є вбудований оператор транзитивного замикання ^, який дозволяє лаконічно виражати рекурсивні обмеження, які в OCL вимагають написання складних рекурсивних функцій.

Як приклад, візьмемо перевірку ациклічності ієрархії класів. OCL вимагає визначення допоміжного методу allParents(), а в той же час в Alloy це можна визначити як

```
fact { no c: Class | c in c.^super }
```

Тобто жоден клас не повинен належати до множини своїх власних предків).

Така архітектура транслятора дозволяє покрити 90% типових структурних обмежень, що використовуються в промисловому моделюванні, забезпечуючи високу точність верифікації без втрати продуктивності.

3.1.2 Формулювання методу

Запропонований метод базується на формальній UML-метамоделі з двома поданнями. Метод орієнтований на інкрементальну локалізовану перевірку консистентності, що забезпечує гарантовану консистентність моделі під час її поступової модифікації.

Визначимо етапи роботи метода.

1. *Виявлення зміни та ідентифікація подання моделі.*

Зміна ідентифікується як така, що стосується одного з подань метамоделі – M_S або M_B. Це визначається на основі структури ідентифікаторів або контексту редагування. Відповідно до структури множин E_S (екземпляри M_S) та E_B (екземпляри M_B), визначається, яка з підмножин зазнала змін, і локалізується відповідний підграф у графі відповідностей $\mu \subset E_S \times E_B$.

2. Синхронізація при зміні структурного подання (JSON → XMI). Якщо змінено структурний елемент, то метод виконує пряму перевірку та синхронізацію впливу на поведінкове подання. Розглянемо підвипадки:

2.1 *Додавання структурного елемента.* При додаванні елемента $s \in E_S$ метод перевіряє, чи виконується правило повноти відповідностей:

$$\forall s \in E_S, s.hasBehavior = true \implies \exists b \in E_B : (s, b) \in \mu. \quad (3.1)$$

Якщо відповідність відсутня, створюється новий поведінковий елемент b, який задовольняє вимогу існування у множині E_B. Це гарантує виконання OCL-інваріанту BehaviorExists.

2.2 *Видалення структурного елемента*. Для кожного b, що має відношення (s,b) $\in \mu$, виконується перевірка на релевантність. Якщо залежність критична, елемент b видаляється, інакше маркується як невалідний. Це підтримує виконання інваріанту AnchorExists.

2.3 *Модифікація структурного елемента*. Метод оцінює, які властивості елемента s були змінені (наприклад, тип, ім’я, атрибути) та виконує часткову перевірку відповідних обмежень консистентності для зв’язаних b, таких що (s,b) $\in \mu$. Приклад перевірки сумісності типів для викликів операцій:

$$\forall m \in Message, m.opRef \neq undefined \implies SignatureMatches(m), \quad (3.2)$$

де SignatureMatches(m) означає, що кількість аргументів у повідомленні m дорівнює кількості параметрів відповідної операції та тип кожного аргументу відповідає або є сумісним з типом відповідного параметра цієї операції.

Це забезпечує дотримання інваріанту SignatureMatches для типів параметрів і викликів.

3. *Синхронізація при зміні поведінкового подання (XMI → JSON).*

3.1 *Додавання поведінкового елемента.* Для кожного нового $b \in E_B$, метод встановлює якір $s \in E_S$ відповідно до вимог інваріанта SingleAnchor:

$$\forall b \in E_B, \exists! \, s \in E_S : (s, b) \in \mu. \tag{3.3}$$

Якщо якір відсутній, формується пропозиція щодо створення або прив'язки відповідного елемента на структурному поданні.

3.2 *Видалення поведінкового елемента.* Перевіряється, чи залишаються структурні елементи, залежні виключно від вилученого елемента поведінки. Наявність таких елементів може свідчити про утворення невалідних структурних фрагментів.

3.3 *Модифікація поведінкового елемента.* Здійснюється локалізована перевірка сумісності параметрів, типів і відповідностей між поданнями. У разі порушення узгодженості формуються рекомендації або пропозиції щодо автоматичного виправлення.

4. *Локалізована перевірка OCL-інваріантів*. Після виконання операцій синхронізації, включаючи можливі автоматичні виправлення, метод здійснює перевірку узгодженості моделі шляхом застосування OCL-інваріантів. Перевірка виконується не для всієї моделі, а лише для множини Affected Elements – елементів, які безпосередньо або опосередковано пов'язані зі зміненим елементом через відношення відповідності μ та інші структурні й поведінкові залежності. Такий підхід дозволяє суттєво скоротити обсяг перевірки та обчислювальні витрати, обмежуючи аналіз лише релевантними фрагментами моделі.

Кожен інваріант OCL має локальний характер і перевіряється на основі даних про суміжні елементи моделі (наприклад, інваріанти класів перевіряються за їх атрибутами та зв'язками, інваріанти повідомлень – за пов'язаними операціями та параметрами). Якщо всі інваріанти виконуються для елементів множини Affected

Elements, зміна вважається прийнятою, а модель повертається у консистентний стан. У разі порушення хоча б одного з інваріантів алгоритм фіксує конфлікт і формує відповідне повідомлення для користувача.

Варто зазначити, що для перевірки складних глобальних властивостей, які неможливо виразити стандартними засобами OCL (наприклад, ациклічності ієрархій успадкування), цей крок валідації передбачає застосування більш потужних формалізмів. Як було обґрунтовано раніше, для таких випадків пропонується двоступеневий підхід, що полягає у трансформації моделі та її обмежень у специфікацію на мові Alloy. Цей процес дозволяє проводити глибоку формальну верифікацію та автоматично генерувати контрприклади для швидкого виявлення причини дефекту. Саме такий підхід був детально продемонстрований раніше на прикладі трансформації інваріанта SingleAnchor у підрозділі 3.1.1.

5. Обробка конфліктів і сповіщення. У разі виявлення залишкових конфліктів система негайно сигналізує про них користувачеві в режимі реального часу. Повідомлення про конфлікт містить опис проблеми (наприклад, «Повідомлення викликає операцію, якої не існує» або «Тип аргументу не відповідає типу параметра операції») та пропонує можливі кроки для усунення: «Створити операцію з такою сигнатурою в класі», «Змінити тип аргументу на правильний» або «Видалити дубльований клас». Користувач може обрати один із запропонованих варіантів, після чого система автоматично вносить відповідні зміни в модель. Після цього інваріанти перевіряються повторно, і якщо конфлікт усунуто, повідомлення зникає.

Якщо автоматичне вирішення конфлікту неможливе (наприклад, коли дубльовані класи мають значні відмінності, і неможливо визначити, який з них слід залишити), система фіксує проблему та передає її на розгляд інженера моделі.

У подальшому викладі використано такі позначення:

– N – загальна кількість елементів UML-моделі;

– k – потужність множини змінених елементів Δ у поточній ревізії, причому $k \ll N$;

– d – максимальна кількість ребер трасування μ, інцидентних кожному елементу. Граф μ зберігається у двобічному хеш-індексі, що забезпечує доступ до

суміжних вершин за O(1).

Інкрементальний алгоритм виконує для кожного $s_i \in \Delta$ повторну перевірку лише локальних інваріантів, а також поширює результат на залежні поведінкові елементи b_j, число яких обмежене величиною d. Таким чином, повна кількість елементарних операцій не перевищує k·d, а за умови сталості d (розріджений граф трасування) асимптотика дорівнює O(k).

Теорема 3.1. Нехай Δ – це множина змінених елементів UML-моделі, |Δ|=k, а d – верхня межа ступеня кожної вершини у графі трасування μ. Тоді час роботи інкрементальної процедури консистентності задовольняє оцінку T(k)=O(k·d).

Доведення. Для кожного елементу si з множини Δ виконується не більше d локальних перевірок інваріантів та одна операція поширення статусу на відповідники bj. Оскільки всі допоміжні структури (індекси, кеші) забезпечують доступ за O(1), сумарна трудомісткість не перевищує k·d; отже, T(k)=O(k·d).

З наведеного випливає, що при k=1 амортизована складність однієї події наближається до O(1), тоді як у загальному разі вона зростає лінійно із числом змін. Повна перевірка моделі, що охоплює всі N елементів, потребує O(N·d) кроків і слугує базовою лінією для порівняння.

Для сценарію, коли вся UML-модель зберігається лише у форматі XMI, локалізована інкрементальна перевірка охоплює одне представлення, і її час позначимо як T1 (k)=O(k·d). У подвійному представленні JSON та XMI перевірка складається з трьох послідовних операцій:

− валідація структурного подання,

− валідація поведінкового подання

− перевірка μ-зв’язків між поданнями.

Перші дві операції мають таку саму верхню оцінку O(k·d), а третя – O(k), оскільки відношення μ підтримується двостороннім хеш-індексом з доступом O(1). Отже, T2(k)=O(k·d+k) = O(k·d).

В результаті, наявність двох форматів не змінює порядку складності порівняно з одноформатною XMI-моделлю; різниця обмежується сталою мультиплікативною надбавкою, що не впливає на масштабованість методу.

Політика методу – максимально автоматизувати виправлення, при цьому забезпечити збереження контролю за моделлю у користувача. Якщо виявлений конфлікт однозначно розв’язний за бізнес-логікою (наприклад, дубль запису – можна видалити один; перейменований клас – слід перейменувати і в усіх діаграмах), то інструмент застосовує виправлення сам і лише повідомляє про нього. У більш тонких випадках користувачу надається вибір. Наприклад, коли знайдено дві дубльовані сутності, система може запропонувати: об’єднати їх, залишити одну (і яку), або продовжити існування обох, але тоді треба вручну їх розрізнити (перейменувати чи уточнити). Система трасує всі зміни, тож історія правок зберігається (можна скасувати автоматичне виправлення, якщо воно виявилося небажаним).

3.2 Метод інкрементальної валідації

Як було показано в аналізі літератури, традиційні методи валідації, реалізовані в більшості сучасних CASE-засобів, покладаються на повну перевірку моделі після внесення будь-яких змін. Такий підхід, хоч і є надійним, характеризується високою обчислювальною затратністю. Як це було показано в попередньому пункті, її складність в загальному випадку оцінюється як O(N), де N – це загальна кількість елементів у моделі. Для великих промислових систем, що можуть містити тисячі елементів, час повної перевірки стає неприйнятно довгим, що робить таку валідацію непрактичною для використання як в ітеративних циклах розробки, так і в конвеєрах безперервної інтеграції.

Фундаментальним недоліком повної валідації є те, що вона ігнорує локальний характер більшості змін. Модифікація одного атрибута чи перейменування однієї операції, як правило, впливає лише на невелику, локалізовану підмножину елементів моделі. Тому виникає потреба у інкрементальній валідації, яка перевіряє лише ті частини поведінкової моделі, які потенційно могли втратити коректність через зміну. По суті, запропонований інкрементальний підхід базується на точному відстеженні та обмеженні поширення впливу будь-якої модифікації.

Інкрементальна валідація передбачає формулювання задачі наступним чином:

для кожної зміни серед елементів поведінкового подання M_B визначити підмножину інваріантів та відповідних елементів, які потребують повторної перевірки, а також залежності з боку елементів структурного подання 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: $M_B \rightarrow M_S$ – відображення елементів M_B на відповідники в M_S.

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

$$Inv^{\Delta} = U_{e \in \Delta MB} Inv(e) \cup Inv\big(\mu(e)\big). \quad (3.4)$$

Тобто до перевірки включаються інваріанти, пов’язані зі зміненими поведінковими елементами, та інваріанти, пов’язані з елементами MS, які з ними консистентні.

Крім цього, для інваріантів з неявною залежністю може знадобитися транзитивне розширення області перевірки за графом залежностей. Тому інструментальна підтримка повинна реалізовувати механізми виявлення залежностей між об'єктами 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 реалізації цього принципу необхідний формальний інструмент, здатний представити складну мережу взаємозв'язків між елементами моделі у вигляді машинооброблюваної структури. Тому для вирішення цієї проблеми вводиться поняття графа залежностей поведінкового подання, що описує відношення впливу між об’єктами у M_B і яка перетворює неявні залежності, розкидані по різних файлах, на явну і зрозумілу структуру.

Нехай поведінкове подання M_B складається з множини елементів:

$$E_{MB} = \{e_1, e_2, \dots, e_n\}, \tag{3.5}$$

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

Визначення. Граф залежностей поведінкового подання – це орієнтований граф у вигляді

$$G_{MB} = (E_{MB}, R), \tag{3.6}$$

де E_{MB} – скінченна множина вершин, що однозначно відповідає множині всіх елементів поведінкового подання моделі (наприклад, станам, переходам, повідомленням, діям тощо), які зберігаються у форматі XMI;

$R \subseteq E_{MB} \times E_{MB}$ – скінченна множина орієнтованих ребер, де ребро $(e_i, e_j) \in R$ представляє відношення залежності: «елемент e_j залежить від елемента e_i». Це означає, що будь-яка зміна в елементі e_i потенційно може призвести до порушення інваріантів, пов'язаних з елементом e_j.

Граф перетворює неявні зв'язки, такі як текстові посилання на ідентифікатори в XMI-атрибутах або тегах профілю, на явні, формальні ребра, що дозволяє системі валідації виконувати швидкий обхід графа для точного визначення зони впливу будь-якої зміни. І в результаті граф стає центральним елементом, з допомогою якого система взаємодіє з моделю, перетворюючи її з набору пасивних даних на активну, запитувану базу знань.

Тобто ту застосовано підхід, що базується на *узагальненні*, де замість обробки окремих випадків, вводиться узагальнена структура – формальний орієнтований граф залежностей. Проте, саме по собі введення графа є лише першим кроком узагальнення. Оскільки не всі залежності є однаково значущими з семантичної точки зору, для побудови ефективного алгоритму валідації необхідний наступний, більш глибокий рівень узагальнення. Цим рівнем є логічна класифікація всієї множини ребер на скінченну кількість фундаментальних типів, що відображають не конкретну реалізацію зв'язку, а його *призначення* в процесі валідації.

У графі залежностей виділяють три логічні різновиди ребер, для виконання більш точного та контекстно-залежного аналіз впливу змін.

Структурні залежності. Цей тип ребер відображає відношення володіння, композиції та агрегації в межах одного подання. Вони фіксують синтаксичну структуру артефактів, описуючи, як одні елементи є контейнерами для інших.

Структурні ребра дозволяють відстежувати каскадні зміни, що стосуються синтаксичної цілісності моделі. Наприклад, видалення класу-контейнера автоматично поширює вплив на всі його атрибути та операції, дозволяючи валідатору перевірити, чи не залишилося «висячих» посилань на ці видалені елементи.

Серед прикладів можна навести наступні:

− ребро від UMLClass до UMLOperation, що моделює відношення owns;

− ребро від UMLPackage до UMLClass, що моделює відношення contains;

− ребро від StateMachine до State (тобто машина станів «містить» стан).

Семантичні залежності. Цей тип ребер фіксує логічні, поведінкові зв'язки між елементами, які визначають функціонування та семантику системи. Вони описують, як елементи взаємодіють один з одним для реалізації певної поведінки.

Семантичні ребра є ключовими для перевірки логічної узгодженості моделі. Вони дозволяють аналізувати вплив семантичних змін. Наприклад, зміна сигнатури операції (зміна її параметрів) поширюється через семантичні ребра на всі повідомлення, що її викликають, дозволяючи валідатору запустити перевірку інваріанта відповідності сигнатур (SignatureMatches) саме для цих повідомлень.

Серед прикладів можна навести наступні:

− ребро від UMLMessage до UMLOperation, що моделює відношення calls;

− ребро від UMLTransition до UMLState, що моделює відношення target;

− ребро від UMLAttribute до UMLClass, що моделює відношення typedBy.

Трасувальні залежності. Цей тип ребер є фізичною, машинооброблюваною реалізацією формального відношення відповідності μ. Вони з'єднують вершини, що належать до різних подань, створюючи міст між структурним та поведінковим світами.

Трасувальні ребра є основою для перевірки міжподаннєвої консистентності. Вони дозволяють поширювати вплив змін між двома фізично розділеними артефактами. Наприклад, перейменування класу в JSON-файлі через трасувальне ребро ініціює перевірку всіх пов'язаних з ним ліній життя в XMI-файлі, забезпечуючи виконання інваріанта LifelineCorrespondence.

Серед прикладів можна навести наступні:

− ребро від UMLLifeline (XMI) до UMLClass (JSON), що моделює відношення represents;

− ребро від UMLTrigger (XMI) до UMLOperation (JSON), що моделює відношення references.

Перевага такого підходу полягає в його здатності до трансформації. Якщо в майбутньому умови зміняться (наприклад, до метамоделі буде додано новий елемент UMLComponent та нове відношення deploys), то не буде потрібно змінювати сам алгоритм валідації. Достатньо лише трансформувати цю нову залежність для конкретного випадку: навчити парсер створювати відповідне ребро в графі G_{MB} та *класифікувати* його як один із трьох узагальнених типів. Алгоритм, що оперує абстракціями, коректно обробить цю зміну, забезпечуючи гнучкість, масштабованість та точне визначення зони впливу.

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 (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:

$$Affected(\delta) = \{e' \in E_{MB} \vee \text{існуєшлях } \delta \to e' \text{у} G_{MB}\}. \quad (3.7)$$

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:

$$Inv^{\Delta} = \cup_{\delta \in \Delta MB} \cup_{e' \in Affected(\delta)} Inv(e'). \quad (3.8)$$

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.

Нижче розглянуто спрощений приклад поведінкового подання M_B, а також побудований для нього граф залежностей між основними елементами подання (рис. 3.1). 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.

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.

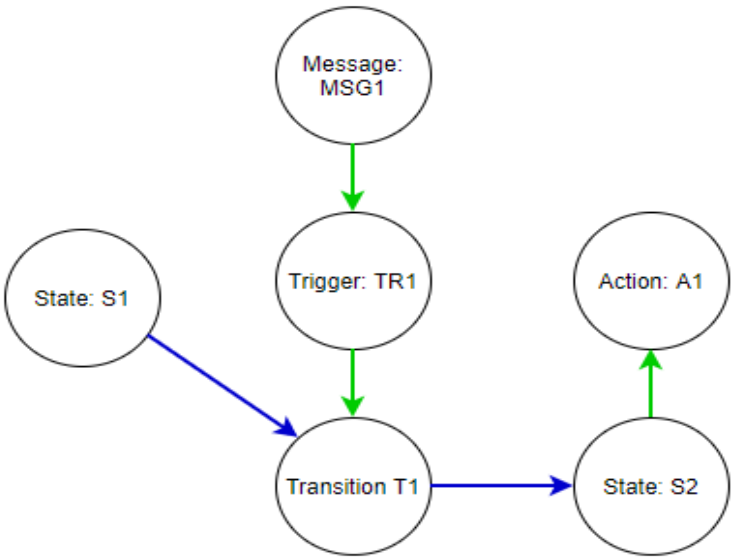

Рисунок 3.1 – Граф залежностей між елементами поведінкової моделі UML

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(ΔMB), 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:

$$\Delta M_B = \{b_1, b_2, \dots, b_k\}, \tag{3.9}$$

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

$$Affected = \cup_{b \in \Delta MB} Affected(b). \tag{3.10}$$

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

$$InvToCheck = \cup_{e \in Affected} Inv(e). \tag{3.11}$$

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:

$$O(k \cdot d), \tag{3.12}$$

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 $O(|M_B|)$.

3.3 Метод двосторонньої синхронізації

Попередній підрозділ обґрунтував метод інкрементальної валідації як обчислювально ефективний інструмент для діагностики неузгодженостей у UML-моделях з двома поданнями. Цей метод є необхідною, але не достатньою умовою для управління архітектурною цілісністю в динамічних середовищах розробки. Валідація за своєю суттю є пасивним, реактивним механізмом: вона виявляє та сигналізує про помилки вже після того, як вони виникли, але не пропонує засобів для їх автоматичного усунення. Для запобігання архітектурному дрейфу необхідний проактивний, корегувальний механізм, здатний не лише знаходити помилки, а й активно підтримувати консистентність моделі в умовах її безперервної еволюції. Тому ця потреба є логічним підґрунтям для розробки методу двосторонньої синхронізації.

Проблема двосторонньої синхронізації формально визначається як задача збереження інваріанта консистентності для моделі з двома поданнями в умовах безперервних та асинхронних змін ΔM_S та ΔM_B. Фундаментальна складність цієї

задачі полягає в тому, що зміни можуть виникати у будь-якому з подань і мати однаково високий пріоритет. Наприклад, розробник може виконати рефакторинг вихідного коду, що призведе до змін у структурному поданні M_S. Одночасно архітектор може уточнити бізнес-логіку шляхом модифікації діаграми послідовності, що спричинить зміни у поведінковому поданні M_B. Центральним викликом є необхідність узгодженого та автоматичного поширення цих змін на відповідне залежне подання, щоб відновити модель до нового, але гарантовано консистентного стану M'.

Тому метою даного підрозділу є розробка методу, що забезпечує автоматичне підтримання узгодженості моделі шляхом поширення змін між її поданнями. Запропонований метод базується на трьох непорушних принципах, які вирішують ключові проблеми, притаманні системам такого класу.

Інкрементальність. Синхронізація повинна бути обчислювально ефективною. Оновленню та повторній перевірці підлягають лише ті елементи моделі, які безпосередньо або транзитивно зачеплені зміною.

Збереження цілісності. Кожна операція синхронізації є атомарною та повинна зберігати консистентність. Процес поширення змін завершується обов'язковим етапом локалізованої верифікації, яка підтверджує, що зачеплена підмножина моделі є узгодженою. У випадку невдачі операція може бути безпечно відкочена, що запобігає внесенню нових помилок та гарантує, що модель ніколи не переходить у некоректний стан.

Керованість. Механізм синхронізації є стабільним та передбачуваним. Він захищений від патології нескінченних циклів за допомогою спеціального механізму маркування походження змін.

У подальшому викладі буде детально розглянуто формальну основу та принципи роботи запропонованого методу. Спочатку буде представлено загальний алгоритм синхронізації, після чого буде проведено глибокий аналіз механізмів поширення змін в обох напрямках: від структури до поведінки ($M_S \to M_B$) та від поведінки до структури ($M_B \to M_S$). Далі буде обґрунтовано механізми, що гарантують цілісність та коректність процесу, включаючи стратегії обробки конфліктів. На

завершення буде представлено механізм уникнення циклів оновлень та проведено порівняльний аналіз запропонованого підходу з існуючими академічними фреймворками.

3.3.1 Формальна основа та теорема інкрементальної консистентності

Перш ніж переходити до побудови механізмів синхронізації між структурним і поведінковим поданнями, необхідно забезпечити верифікацію такої UML-моделі як передумову її коректного функціонування. У цьому контексті під верифікацією розуміється підтвердження відповідності моделі заздалегідь визначеним валідаційним правилам, що формалізують допустимі структури та поведінкові патерни відповідно до семантики UML.

Структурна частина моделі представлена у форматі JSON відповідно до запропонованої метамоделі. Валідаційні правила для цієї частини були раніше систематизовані у розділі 2, зокрема обов'язковість імен для сутностей, коректність типізації атрибутів і зв'язків, єдність ідентифікаторів у межах одного простору імен, відповідність класів, атрибутів і операцій їх метатипам. Ці правила інтерпретуються як множина предикатів σ_1, σ_2, …, σ_n, кожен з яких перевіряється над множиною елементів M_S.

Формальна модель валідації виглядає наступним чином:

$$\forall m \in M_S : i = 1 n \sigma_i(m) = true. \quad (3.13)$$

Поведінкова частина представлена у форматі XMI, який реалізує специфікацію UML 2.5.x з розширенням у вигляді спеціального профілю. Валідаційні інваріанти для поведінкових діаграм також наведені в розділі 2 і охоплюють діаграми станів (перевірка умов переходу, унікальності початкового стану, наявності переходів), діаграми діяльності (коректність потоків керування, відповідність типів вхідних/вихідних даних) та діаграми послідовностей (зв'язність між повідомленнями, послідовність активацій, консистентність із Lifeline).

Нехай ψ_1, ψ_2,…,ψ_k – предикати валідації поведінки над множиною M_B. Тоді модель задовольняє поведінкову верифікацію, якщо:

$$\forall b \in M_B : j = 1k\psi_J(b) = true. \quad (3.14)$$

Ключовою вимогою використання цих двох подань є семантична відповідність між структурою та поведінкою. Вона реалізується через відображення μ, яке визначає, які елементи M_B мають бути пов'язані з відповідними елементами M_S. Правила відповідності включають:

− наявність посилань jsonRef, triggerSource в елементах M_B;

− відповідність типів дій, викликів або повідомлень до відповідних методів або сигналів у структурі;

− відповідність параметрів і типів даних між InputPin та структурними атрибутами або операціями.

Тому, модель вважається верифікованою, якщо M_S задовольняє всі структурні інваріанти, M_B задовольняє всі поведінкові інваріанти, а також якщо всі елементи M_B, що мають трасувальні посилання, відображаються на коректні елементи M_S і виконуються узгоджувальні правила.

Ця верифікована база виступає точкою відліку для всіх подальших інкрементальних оновлень і є необхідною умовою запуску двосторонньої синхронізації, яка не повинна порушувати верифікованість моделі після застосування змін.

Далі, опираючись на вищевказаний формальний опис UML-моделі, необхідно сформувати теорему інкрементальної консистентності такої UML-моделі з двома поданнями.

Теорема 3.2. Нехай існує наступна UML-модельm, яка описується формулою 2.1.

Також нехай:

Σ – сукупність формальних обмежень консистентності у вигляді набору OCL-інваріантів, які визначають допустимість моделей;

$\Delta \subseteq M_S \cup M_B$ – множина елементів, що зазнали змін внаслідок інкрементального редагування;

Affected(Δ,μ) $\subseteq M_S \cup M_B$ – замикання змінених елементів через відношення відповідності μ, тобто множина всіх елементів, для яких існує зв'язок із елементами з Δ або суміжність за структурними/поведінковими відношеннями.

Тоді, якщо після зміни елементів із Δ усі інваріанти з Σ виконуються для кожного елемента з множини Affected(Δ,μ), то модель M загалом також задовольняє всі інваріанти з Σ, тобто

$$\left(\forall e \in Affected(\Delta, \mu): M \vDash \Sigma(e)\right) \Rightarrow M \vDash \Sigma. \tag{3.15}$$

Доведення від супротивного. Припустимо протилежне твердженню теореми: нехай після змін у моделі виконується

$$\forall e \in Affected(\Delta, \mu): M \vDash \Sigma(e), \tag{3.16}$$

але вся модель неконсистентна, тобто:

$$\exists \sigma \in \Sigma: M \nvDash \sigma. \tag{3.17}$$

За визначенням, кожен інваріант σ формується локально, тобто його істинність залежить лише від обмеженої множини елементів:

$$Context(\sigma) \subseteq M_S \cup M_B. \tag{3.18}$$

З того, що M ⊭ σ, випливає, що хоча б один з елементів контексту Context(σ) порушує вимогу цього інваріанту.

Але якщо інваріант σ був порушений, і цей інваріант залежить від деякого елемента e, тоді або e ∈ Δ, тобто був змінений безпосередньо, або e ∈ Affected(Δ,μ), тобто зміни торкнулися суміжного елемента.

Це суперечить нашому припущенню, що всі інваріанти виконуються на Affected(Δ,μ).

Отже, припущення про те, що M ⊭ Σ при M⊨Σ(e) ∀e∈Affected(Δ,μ), є хибним.

В результаті, ця теорема дозволяє обмежити обчислювальну перевірку консистентності лише підмножиною Affected(Δ,μ), що істотно знижує складність верифікації в умовах інкрементального редагування.

Будь-яка зміна в моделі представляється у вигляді множини змін Δ, яка може бути декомпозована на дві підмножини: ΔM_S, що містить зміни у структурному поданні, та ΔM_B, що містить зміни у поведінковому поданні.

Центральною ідеєю методу є концепція поширення змін, яка базується на відношенні відповідності μ та графі залежностей G. Будь-яка зміна $\delta s \in \Delta M_S$, що стосується елемента $s \in M_S$, ініціює пошук множини залежних поведінкових елементів $Bs=\{b|(s,b)\in\mu\}$. Зміна δs трансформується у відповідну множину змін {δb}, які застосовуються до елементів з Bs. Аналогічна логіка застосовується у зворотному напрямку для зміни $\delta b\in\Delta M_B$. Граф залежностей G використовується для ідентифікації та поширення другорядних ефектів, що виходять за межі прямого зв'язку, визначеного відношенням μ. Цей механізм гарантує, що зміна, внесена в одному поданні, буде коректно та повно відображена в іншому, підтримуючи цілісність моделі.

Загальний алгоритм синхронізації є високорівневим процесом, що складається з чотирьох послідовних кроків, які забезпечують кероване та верифіковане поширення змін.

Крок 1. Локалізація змін. Система ідентифікує початкову множину змін Δ та розподіляє її на дві підмножини: ΔM_S та ΔM_B. Кожна зміна позначається маркером походження, що є критично важливим для подальшої обробки. Формально:

$$\Delta = \Delta M_S \cup \Delta M_B. \tag{3.19}$$

Далі виконується визначення та побудова зачеплених елементів. На основі початкових змін ΔM_S та ΔM_B система використовує граф залежностей G та

відношення відповідності μ для обчислення транзитивного замикання впливу. Результатом є множина Affected, що містить усі елементи моделі, які були або безпосередньо змінені, або залежать від змінених елементів. Формально:

$$Affected = x \in \Delta Affected(x). \quad (3.20)$$

Цей крок дозволяє точно локалізувати область, що потребує уваги, реалізуючи принцип інкрементальності.

Крок 2. Пропагування змін. Для кожного зміненого елемента виконується спрямоване оновлення пов'язаних об'єктів. Якщо зміна відбулася в M_S, оновлюються або створюються відповідні елементи в M_B (наприклад, клас → Lifeline, атрибут → InputPin). Якщо зміна в M_B, виконується оновлення відповідних елементів M_S (наприклад, Transition → булевий атрибут, Message → операція або сигнал). При цьому в обох випадках актуалізуються трасувальні зв'язки μ.

Результатом цього кроку є оновлений стан моделі, який потенційно є консистентним.

Крок 3. Локальна верифікація. На основі теореми інкрементальної консистентності перевіряються лише інваріанти, які залежать від елементів із множини Affected. Якщо всі локальні перевірки успішні, модель вважається узгодженою. У протилежному випадку зміна відхиляється або маркується як така, що потребує доопрацювання.

Крок 4. Оновлення відношення відповідності. У випадку успішної верифікації та якщо в процесі пропагації були створені або видалені елементи, система оновлює відношення μ, додаючи або видаляючи відповідні пари (s, b), щоб відобразити новий стан моделі.

Запропонований метод володіє кількома ключовими властивостями, що забезпечують його ефективність та достовірність. По-перше, він є *інкрементальним*, оскільки перевірці підлягає лише та частина моделі, яку зачепили локальні зміни, що значно зменшує обчислювальну складність. По-друге, гарантується його *завершуваність*, тобто метод завжди зупиняється, оскільки він не створює

рекурсивного розширення множини змін. Крім того, для *уникнення циклів* нескінченної синхронізації застосовуються спеціальні маркери походження змін (Push або Pull). Нарешті, метод підтримує *семантичну консистентність*: поведінкові елементи, створені внаслідок структурних змін, зберігають повну відповідність до типів, параметрів і залежностей, визначених у структурному поданні M_S.

3.3.2 Деталізація кроку пропагації

Процес поширення змін від структурного подання до поведінкового є ключовим для підтримки узгодженості моделі при рефакторингу вихідного коду. Розглянемо реакцію системи на три основні типи змін.

Додавання елемента $s \in M_S$. Коли розробник додає новий структурний елемент (наприклад, клас або операцію), система аналізує його властивості. Якщо для нового елемента s встановлено прапорець hasBehavior=true, це означає, що він повинен мати відповідне поведінкове представлення. Щоб негайно задовольнити інваріант BehaviorExists, система автоматично створює «заглушку» (placeholder) у поведінковому поданні M_B. Наприклад, для нового класу це може бути порожня діаграма станів, а для нової операції – шаблонна дія (Action) або повідомлення (Message). Одночасно створюється новий поведінковий елемент b, і до відношення відповідності μ додається нова пара (s,b). Цей механізм гарантує, що модель не переходить у неповний стан і надає архітектору готовий каркас для подальшої деталізації поведінки.

Видалення елемента $s \in M_S$. Видалення структурного елемента є потенційно руйнівною операцією, що вимагає ретельної обробки залежностей. Система знаходить усі поведінкові елементи $\{b_i\}$ такі, що $(s,b_i) \in \mu$. Оскільки видалення s порушує інваріант AnchorExists для всіх залежних b_i, система ініціює стратегію вирішення конфлікту. Залежно від налаштувань та контексту, вона може або автоматично видалити всі залежні елементи b_i (каскадне видалення), або, що є більш безпечним підходом, позначити їх як невалідні, що мають «висяче посилання». В останньому випадку система інформує користувача про виникнення неузгодженості

та пропонує варіанти вирішення: підтвердити видалення залежних діаграм або скасувати видалення початкового елемента s.

Модифікація елемента $s \in M_S$. Реакція на модифікацію залежить від характеру зміни.

Якщо змінюється ім'я елемента s, система автоматично поширює цю зміну на імена всіх пов'язаних елементів $b \in M_B$. Це проста, але важлива операція для підтримки читабельності та узгодженості моделі.

Також можлива зміна сигнатури операції. Це найважливіший та найскладніший випадок. Припустимо, розробник змінює сигнатуру операції $o \in M_S$ (наприклад, додає новий параметр). Система ідентифікує всі повідомлення (Message) $m \in M_B$, які моделюють виклик цієї операції (тобто, для яких $(o,m) \in \mu$). Для кожного такого повідомлення m система повторно запускає перевірку інваріанта SignatureMatches. Якщо виявлено невідповідність (наприклад, у повідомленні відсутній новий обов'язковий параметр), ця невідповідність позначається як семантична помилка високого пріоритету. Система повідомляє користувача про всі місця в поведінкових діаграмах, які стали недійсними і потребують оновлення.

Синхронізація у зворотному напрямку є не менш важливою, оскільки дозволяє архітекторам та аналітикам розвивати модель з точки зору поведінки, гарантуючи, що структурна частина буде відповідати цим змінам.

Додавання елемента $b \in M_B$. Коли архітектор додає новий поведінковий елемент, що вимагає структурного якоря (наприклад, лінія життя Lifeline у діаграмі послідовності), система перевіряє наявність посилання на відповідний структурний елемент (наприклад, через атрибут jsonRef). Якщо таке посилання відсутнє, це є порушенням інваріанта AnchorExists. Система негайно сигналізує про помилку та пропонує користувачеві або прив'язати новий поведінковий елемент до існуючого структурного елемента (наприклад, обрати клас зі списку), або ініціювати створення нового структурного елемента «на льоту».

Видалення елемента $b \in M_B$. Видалення поведінкового елемента, як правило, є менш руйнівною операцією. Однак, якщо видалений елемент b був останнім поведінковим представленням для структурного елемента s, який має на собі

прапорець hasBehavior=true, це створює ситуацію, коли структурний елемент декларує наявність поведінки, але фактично її не має. У цьому випадку система може запропонувати користувачеві виконати рефакторинг: змінити прапорець s.hasBehavior на false, щоб привести модель у більш точний та узгоджений стан.

Модифікація елемента $b \in M_B$. Зміни в поведінкових елементах часто вимагають відповідних змін у структурі. Наприклад, якщо аналітик додає новий параметр до повідомлення (Message) m у діаграмі послідовності, система знаходить відповідну операцію o $\in M_S$ через відношення μ. Далі вона аналізує, чи можна безпечно оновити сигнатуру операції o. Якщо доданий параметр може бути інтерпретований як необов'язковий або має тип, що легко виводиться, система може автоматично оновити M_S. Однак, якщо зміна є несумісною, то система фіксує конфлікт і повідомляє користувача про неможливість автоматичної синхронізації, вимагаючи ручного втручання для вирішення неузгодженості.

3.3.3 Стратегія наскрізного збереження контексту

Критичною проблемою існуючих засобів трансформації моделей є втрата інформації, яка не входить до стандарту метамоделі UML, але є необхідною для практичного використання інструментарію. До такої інформації належать координати візуалізації та специфічні розширення CASE-засобів. Для вирішення цієї проблеми у дисертаційній роботі розроблено та формалізовано стратегію наскрізного збереження контексту.

Введемо розширене визначення елемента моделі. Нехай M – множина елементів моделі. Кожен елемент e ∈ M представимо у вигляді кортежу:

$$e = < S, C > \tag{3.21}$$

де S – семантична складова, що визначає структуру та поведінку системи, C – контекстна складова, яка не впливає на логіку виконання, але визначає подання моделі в інструментальному середовищі.

Структуру контексту визначаємо як

$$C = < L, V_{meta} > \quad (3.22)$$

де L – параметри візуалізації (координати {x, y}, розміри {w, h}, кольори, точки вигину ліній), а V – пропрієтарні розширення CASE-засобу.

Основою запропонованої стратегії є принцип ортогональності, який постулює незалежність процесів валідації семантики від контекстних даних. Формально, функція валідації Val та функція трансформації Trans визначаються так як

$$Val(< S, C >) \equiv Val(S) \quad (3.23)$$

та

$$\boldsymbol{Trans}(< \boldsymbol{S}, \boldsymbol{C} >) \rightarrow < \boldsymbol{S'}, \boldsymbol{C'} > \quad (\mathbf{3.24})$$

Метою стратегії є забезпечення умови C' ≈ C (збереження контексту) при змінах S → S', що не впливають на топологію відображення. Для реалізації принципу ортогональності розроблено архітектурний патерн, який трактує контекст C як непрозорий об'єкт під час обробки. Алгоритм складається з трьох фаз:

1. Вилучення. Під час парсингу вихідного XMI-файлу, парсер відокремлює стандартні вузли UML (що формують S) від вузлів розширень (що формують C). Об'єкт Blob(C) зберігається у пам'яті чи іншому середовищі як серіалізований фрагмент XML-дерева (наприклад, об'єкт lxml.Element), без спроби інтерпретації його вмісту.

2. Збереження. Під час маніпуляцій з моделлю операції виконуються виключно над S. Blob(C) залишається асоційованим з унікальним ідентифікатором елемента (xmi:id).

3. Ін’єкція. При генерації цільового файлу відбувається реінтеграція контексту. Для кожного елемента e_{new} застосовується стратегія злиття. Нова конфігурація C_{new}

визначається залежно від стану елемента. Якщо ідентифікатор нового елемента e_{new} збігається з ідентифікатором старого елемента e_{old}, то конфігурація не змінюється і залишається рівною C_{old}. Якщо ж e_{new} є новим елементом, то для нього генерується нова конфігурація.

Цей підхід забезпечує ідемпотентність операції: Serialize(Parse(File)) ≡ File (з точністю до форматування), що усуває проблему так званого шуму в системах контролю версій.

Для обробки сценарію додавання нових елементів (коли C_{old} відсутній), розроблено евристичний алгоритм для інкременатального авто-лейаута. Оскільки нові елементи, створені у текстовому редакторі, не мають координат, їх наївне розміщення у точці (0,0) робить діаграму нечитабельною. Тому запропонована функція використовує відносне позиціонування:

Першим етапом є визначення якоря, де спершу знаходиться батьківський елемент P або останній доданий сусідній елемент N. Після чого виконується розрахунок зміщення. Для цього нові координати (x', y') обчислюються як:

$$x' = x_{anchor} + \Delta_x + random(\varepsilon) \tag{3.25}$$

та

$$y' = y_{anchor} + \Delta_y + random(\varepsilon) \tag{3.26}$$

де Δ – фіксований крок сітки, а ε – невелике випадкове відхилення для запобігання повному накладанню блоків при масовій генерації.

Закінчується це генерацією XML-контейнера, в рамках якого створюється мінімально необхідний дескриптор візуалізації, сумісний з цільовим CASE-засобом. При цьому треба зазначити, що алгоритм авто-лейаута додатково перевіряє bounding box існуючих елементів, щоб уникнути повного перекриття, розміщуючи нові елементи у найближчій вільній клітинці віртуальної сітки.

Таким чином, застосування стратегії наскрізного збереження контексту дозволяє досягти повної сумісності розробленого інструментарію з промисловими стандартами (XMI 2.x) та пропрієтарними середовищами (MagicDraw, Enterprise Architect), усуваючи бар'єри втрати візуальних данних або специфічних та пропрієтарних елементів.

3.3.4 Забезпечення безпечної синхронізації та інтеграції з VCS

Критичним аспектом розробки автоматизованих засобів MDE є вирішення проблеми незворотності змін. Агресивні алгоритми синхронізації, що діють за принципом, де або застосовується остання зміна, або автоматично видаляють «висячі» посилання для забезпечення цілісності, можуть призвести до втрати інтелектуальної власності. У контексті промислової розробки, де вартість відновлення моделі є високою, така поведінка інструменту є неприпустимою.

Тому для мінімізації цих ризиків у роботі було формалізовано та впроваджено політику ненавмисної деструктивності.

Визначення. Будь-яка автоматизована операція трансформації T: M → M', що призводить до зменшення інформаційної ентропії системи (таких як видалення елементів, атрибутів або зв'язків), заборонена для виконання у фоновому режимі та вимагає явного підтвердження з боку оператора. Тому на основі цього принципу розроблено таксономію змін, яка визначає поведінку механізму синхронізації.

Адиктивні зміни. Це зміни, за яких $M_{old} \subset M_{new}$. Сюди належать: створення нових класів, додавання атрибутів, розширення списку літералів перелічень. При цих змінах виконується автоматичне застосування, де система виконує синхронізацію миттєво, оскільки ризик втрати даних відсутній.

Деструктивні зміни. Це зміни, що передбачають видалення елементів ($e \in M_{old}$, $e \notin M_{new}$) або звуження типів даних. В даних випадках відбувається блокування та валідація. Тобто автоматична синхронізація призупиняється, після чого генерується diff-звіт, що підсвічує елементи, які підлягають видаленню. Користувач повинен

обрати дію: Confirm Deletion (підтвердити видалення) або Restore (відновити елемент у джерелі).

Окремим класом проблем є інтеграція модельних інструментів із системами контролю версій. Тексто-орієнтована природа алгоритмів злиття, яка є ефективною для вихідного коду, виявляється недостатньою для графових структур даних, якими є UML-моделі (навіть у форматі JSON).

В результаті було ідентифіковано артефактів, що виникають після автоматичного злиття гілок, які є валідними синтаксично, але порушують семантичні правила метамоделі. Це може виглядати наступним чином::

– Гілка A. Розробник видаляє клас User.

– Гілка B. Розробник додає новий атрибут email до класу User.

– Результат злиття. Git, не розуміючи семантики, може об'єднати ці зміни так, що атрибут email залишиться у підвішеному стані у файлі без батьківського класу, або клас буде відновлено частково.

Оскільки зміна внутрішніх алгоритмів Git виходить за межі дослідження, запропоновано архітектурне рішення на рівні процесу CI/CD – стратегія перевірки узгодженості після злиття. Вона полягає у використанні механізму Git Hooks (зокрема, post-merge або pre-commit на етапі злиття) для запуску полегшеної версії валідатора моделі.

Алгоритм валідації передбачає декілька етапів. Спершу виконується детекція змін, в рамках чого хук перевіряє, чи торкнулося злиття файлів моделей (.json, .xmi). Якщо так, то далі запускається консольна утиліта, яка завантажує об'єднану модель у пам'ять і перевіряє базові структурні інваріанти (таких як наявність усіх типів, коректність посилань і тд). Якщо в результаті перевірки були виявлення невалідні артефакти, (наприклад, посилання на неіснуючий ID), то коміт злиття блокується з характерним повідомленням про помилку. Якщо ж модель консистентна, процес завершується успішно.

Такий підхід дозволяє доповнити стандартні можливості Git семантичним аналізом, запобігаючи потраплянню пошкоджених моделей до основного репозиторію, що є критичним для забезпечення безперервності процесу розробки.

3.4 Висновки до розділу 3

У розділі розроблено цілісний науково-методичний апарат для динамічного управління консистентністю UML-моделі з двома поданнями. Цей апарат є логічним розвитком статичної метамоделі, запропонованої в попередньому розділі, та забезпечує перехід від теоретичного опису до практичних механізмів підтримки архітектурної цілісності в умовах безперервної еволюції моделі.

Розроблено метод автоматизованого забезпечення консистентності, що базується на системі формальних обмежень, виражених мовою OCL, для верифікації семантичної коректності та логічної узгодженості моделі. Для подолання теоретичних обмежень OCL, зокрема нездатності виражати транзитивне замикання, запропоновано двоступеневий підхід. Він полягає у трансформації моделі та її обмежень у специфікацію на мові Alloy, що дозволяє проводити глибоку формальну верифікацію та генерувати контрприклади за допомогою потужних SAT-вирішувачів.

Створено обчислювально ефективний метод інкрементальної валідації, що вирішує проблему високої затратності повної валідації. Ключовою інновацією методу є введення формального графа залежностей з класифікованими ребрами (структурними, семантичними та трасувальними). Цей граф дозволяє точно локалізувати область впливу будь-якої зміни та застосовувати перевірку лише до множини зачеплених елементів, що знижує асимптотичну складність валідації з $O(N)$ до $O(k \cdot d)$.

Розроблено проактивний метод двосторонньої синхронізації, який не лише діагностує помилки, а й автоматично поширює зміни між структурним та поведінковим поданнями для підтримки їхньої узгодженості в реальному часі. Достовірність та коректність цього методу математично обґрунтована сформульованою та доведеною теоремою інкрементальної консистентності, яка гарантує, що локалізованої перевірки на множині зачеплених елементів достатньо для підтвердження глобальної консистентності всієї моделі.

Запропоновані три методи утворюють єдиний, багаторівневий та взаємопов'язаний науково-методичний апарат. Цей апарат забезпечує як

реактивну діагностику (виявлення неузгодженостей за допомогою валідації), так і проактивну корекцію (автоматичне усунення розбіжностей шляхом синхронізації), створюючи комплексну систему управління цілісністю моделі.

В результаті, розроблений у розділі апарат є завершеною теоретичною та алгоритмічною основою для створення практичного інструментального засобу. Він вирішує ключову проблему підтримки архітектурної цілісності в динамічних умовах сучасної ітеративної розробки, а його ефективність та практична реалізовність будуть показані та експериментально підтверджені в наступному розділі дисертації.

Матеріали розділу опубліковані в роботах [106, 161, 162, 165, 166].

4 АРХІТЕКТУРА ТА ЕКСПЕРИМЕНТАЛЬНА ВАЛІДАЦІЯ ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ

4.1 Архітектура програмного плагіна для інкрементальної валідації

Розробивши в попередніх розділах повний науково-методичний апарат, що включає методи представлення моделі, інкрементальної валідації, двосторонньої синхронізації та трансформації, завершальним етапом дослідження є їхня практична імплементація та експериментальна валідація в рамках єдиного інструментального засобу. Цей розділ присвячено опису архітектури, реалізації та тестуванню розробленого програмного плагіна, який слугує доказом того, що запропоновані теоретичні концепції є не лише коректними, а й практично реалізовними та обчислювально ефективними. Даний підрозділ починає цей опис з обґрунтування загальних архітектурних рішень, детального опису кожного модуля та його відповідальності в програмі, і на завершення –представлення загального потоку даних у системі.

Архітектура розробленого плагіна не є довільною, а є прямим, фізичним втіленням формальної метамоделі, розробленої в розділі 2. Цей зв'язок є основою усього проектувального рішення. Розділення плагіна на модулі, що оперують з артефактами у форматах JSON та XMI, безпосередньо випливає з концептуального розділення метамоделі два подання. Це гарантує, що програмна реалізація точно відповідає теоретичній основі роботи.

В основі архітектури лежить принцип розділення відповідальності, що реалізовано через модульну структуру, що забезпечує високу зв'язність всередині кожного модуля та низьке зчеплення між ними. Це, в свою чергу, гарантує легкість тестування, супроводжуваності та, що найголовніше, майбутньої розширюваності системи, що теж є ключивими вимогами для науково-дослідних прототипів.

Центральним архітектурним елементом, що забезпечує низьке зчеплення та високу гнучкість, є *внутрішня модель*. Це не просто структура даних, а стратегічний механізм роз'єднання, реалізований як набір об'єктів у пам'яті, що слугують

канонічним, незалежним від формату представленням UML-моделі. Внутрішня модель діє як посередник, що повністю ізолює логіку аналізу вихідного коду від логіки валідації та генерації цільових артефактів. Така архітектура аналогічна класичній трикомпонентній структурі компіляторів («фронт-енд – проміжне подання – бек-енд»), що надає їй додаткової академічної строгості. Модулі-аналізатори (parsers) виступають як «фронт-енд», трансформуючи синтаксичні конструкції різних мов програмування у єдине проміжне подання – внутрішню модель. Модулі-генератори (generators) діють як «бек-енд», серіалізуючи це подання у різні цільові формати. Ядро системи (core) та валідатори (validators) оперують виключно з цим стабільним проміжним поданням, нічого не знаючи про специфіку вхідних чи вихідних форматів. Разом це перетворює плагін з вузькоспеціалізованого інструменту на універсальний та розширюваний валідаційний рушій. Для додавання підтримки нової мови програмування, наприклад Java, достатньо реалізувати лише новий модуль-аналізатор, не вносячи жодних змін до ядра системи, валідаторів чи генераторів.

Відповідно до принципу модульності, архітектура плагіна декомпозована на п'ять функціональних модулів, кожен з яких має чітко визначену зону відповідальності.

Модуль parsers (Аналізатори). Цей модуль відповідає за перший етап роботи системи – аналіз вихідного коду та його перетворення на об'єкти внутрішньої моделі. На поточному етапі дослідження реалізовано парсер для мови програмування Python. Він використовує вбудований у стандартну бібліотеку модуль ast для побудови абстрактного синтаксичного дерева (abstract syntax tree, AST) з вихідного коду. Спеціалізований клас-відвідувач, що успадковує ast.NodeVisitor, рекурсивно обходить це дерево, ідентифікуючи ключові структурні елементи – класи, їхні атрибути, методи та відношення успадкування. На основі цієї інформації створюються та наповнюються відповідні екземпляри dataclasses для внутрішньої моделі.

Діаграма класів цього модуля зображена на рис. 4.1.

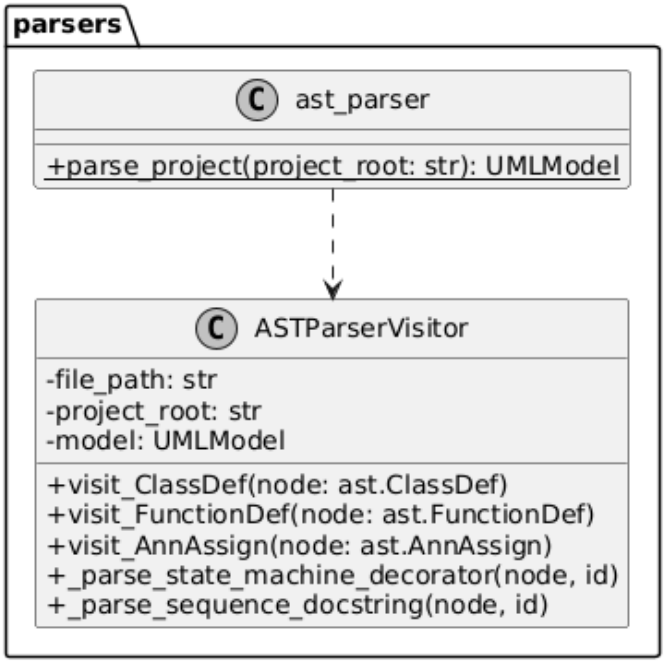

Рисунок 4.1 – Діаграма класів модуля *parsers*

Модуль core (Ядро системи). Ядро системи відповідає за зберігання та управління внутрішнім станом моделі. Цей модуль є центральним сховищем даних та логіки, що об'єднує інші компоненти, і складається з трьох ключових елементів:

– **Визначення внутрішньої моделі.** Набір dataclasses мови Python, що точно відображають метасутності UML (наприклад, UMLClass, UMLOperation, UMLAttribute), визначені в рамках формалізації структурного подання M_S.

– **Реалізація графа залежностей G_{MB}.** Практична імплементація орієнтованого графа залежностей. Граф будується в пам'яті одночасно з наповненням внутрішньої моделі. Його вершинами є всі елементи моделі, а ребра класифікуються на три типи – структурні, семантичні та трасувальні, що дозволяє виконувати точний аналіз впливу змін.

– **Логіка персистентності.** Механізми для збереження та завантаження проміжних артефактів. Ця функціональність є важливою для кешування результатів аналізу та оптимізації інкрементальних операцій при повторних запусках.

Діаграма класів цього модуля зображена на рис. 4.2.

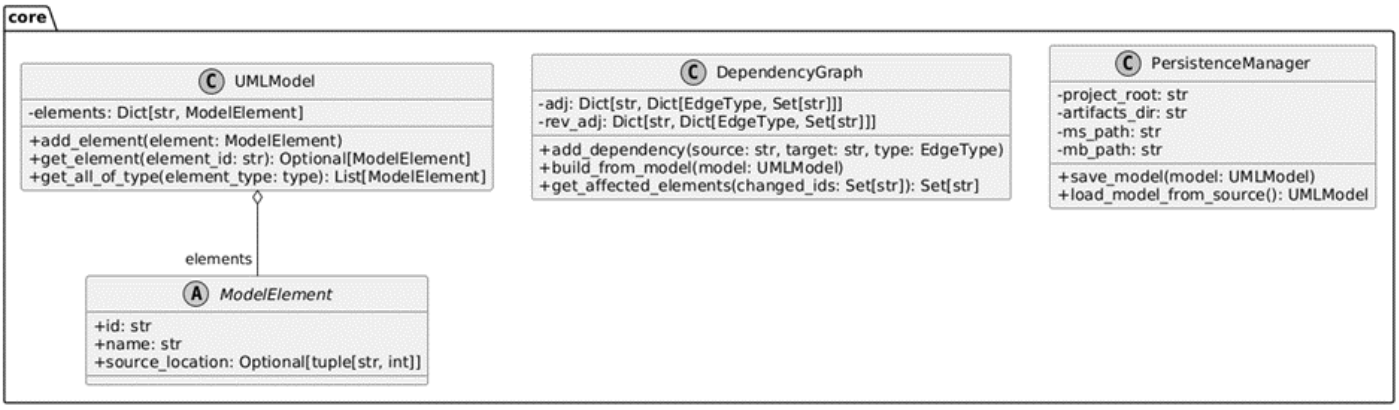

Рисунок 4.2 – Діаграма класів модуля *core*

Модуль generators (Генератори). Цей модуль виконує функцію «бек-енду» системи, відповідаючи за серіалізацію об'єктів внутрішньої моделі у цільові файлові артефакти. Він отримує на вхід готову внутрішню модель з модуля core і генерує два файли, що разом складають повну UML-модель з двома поданнями.

Перший файл – це model.struct.json. Цей файл містить структурне подання моделі і генерується у повній відповідності до уніфікованої JSON Schema. Це забезпечує перший, найшвидший рівень валідації – синтаксичний.

Другий файл – це model.behav.xmi. Файл, що містить поведінкове подання. Його генерація відбувається з урахуванням спеціалізованого UML-профілю, розробленого та описаного в розділі 2. У XMI-документ вбудовуються необхідні трасувальні метадані (наприклад, стереотип «JsonRef»), які є фізичною реалізацією відношення відповідності μ і зв'язують поведінкові елементи з їхніми структурними аналогами в JSON-файлі.

Діаграма класів цього модуля зображена на рис. 4.3.

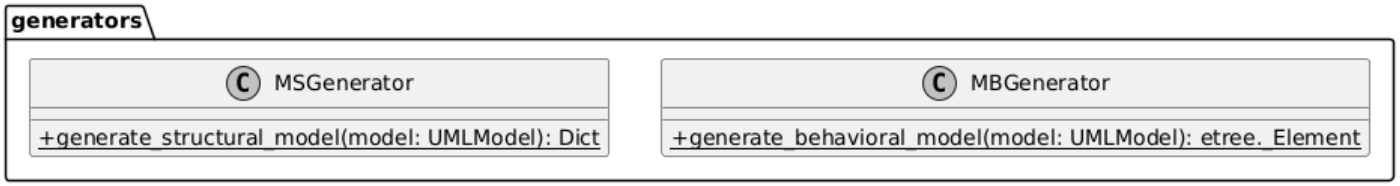

Рисунок 4.3 – Діаграма класів модуля *generators*

Модуль validators (Валідатори). Цей модуль є інтелектуальним ядром плагіна,

що реалізує всі рівні перевірки консистентності. Його архітектура втілює принцип «fail-fast», створюючи багаторівневий валідаційний конвеєр зі зростаючою обчислювальною вартістю та семантичною глибиною. Цей підхід дозволяє оптимізувати баланс між продуктивністю та повнотою аналізу. Прості синтаксичні помилки виявляються миттєво на етапі генерації, поширені семантичні неузгодженості – швидко за допомогою інкрементального аналізу, а глибокі архітектурні дефекти – за допомогою формальної верифікації.

Сам модуль складається з двох основних компонентів.

− **Валідаційний рушій (ValidationEngine).** Це компонент, що реалізує високоефективний інкрементальний алгоритм ValBeh. Рушій використовує граф залежностей G_{MB} з модуля *core* для точної локалізації області перевірки, аналізуючи лише ті елементи, які були зачеплені останньою зміною.

− **Міст до Alloy (AlloyBridge).** Це компонент, що реалізує підхід глибокої формальної верифікації. Він транслює підмножину внутрішньої моделі та її обмежень у специфікацію мови Alloy. Потім, за допомогою зовнішнього SAT-вирішувача, виконується вичерпний пошук контрприкладів, що дозволяє виявляти складні транзитивні дефекти, такі як цикли в ієрархії успадкування, які недоступні для простих локальних перевірок.

Діаграма класів цього модуля зображена на рис. 4.4.

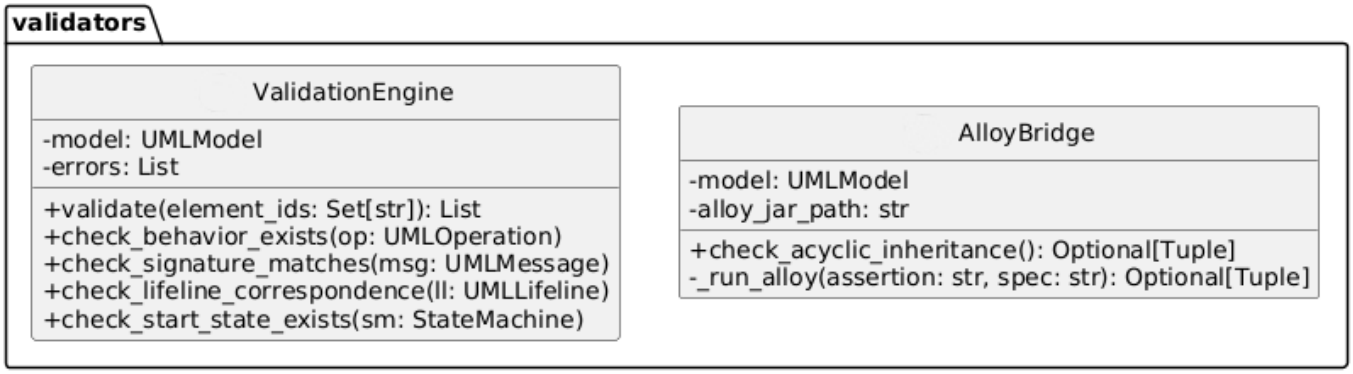

Рисунок 4.4 – Діаграма класів модуля *validators*

Модуль integration (Інтеграція). Цей модуль слугує зовнішнім фасадом системи та надає інтерфейси для взаємодії з плагіном. По суті, він є стратегічним

компонентом, що будує методологічний міст між світом модельно-орієнтованої інженерії та світом сучасних інженерних практик Agile/DevOps, усуваючи розрив, обґрунтований у розділі 1. Замість того, щоб змушувати розробників переходити у відокремлені CASE-засоби, архітектура плагіна приносить потужність формальних методів безпосередньо в їхні звичні інструменти.

В ньому було реалізовано два основні інтерфейси.

− **Інтерфейс командного рядка (CLI).** Надає можливість запускати аналіз та валідацію з командного рядка. Це є необхідною умовою для інтеграції плагіна в автоматизовані конвеєри CI/CD, такі як Jenkins або GitHub Actions, перетворюючи валідацію моделі на невід'ємну частину процесу збірки та тестування.

− **Програмний інтерфейс (API).** Надає програмний інтерфейс для взаємодії з функціональністю плагіна. Цей API є основою для створення розширень до популярних IDE, наприклад, для Visual Studio Code, що дозволяє забезпечити миттєвий зворотний зв'язок розробнику безпосередньо в процесі написання коду.

Діаграма класів цього модуля зображена на рис. 4.5.

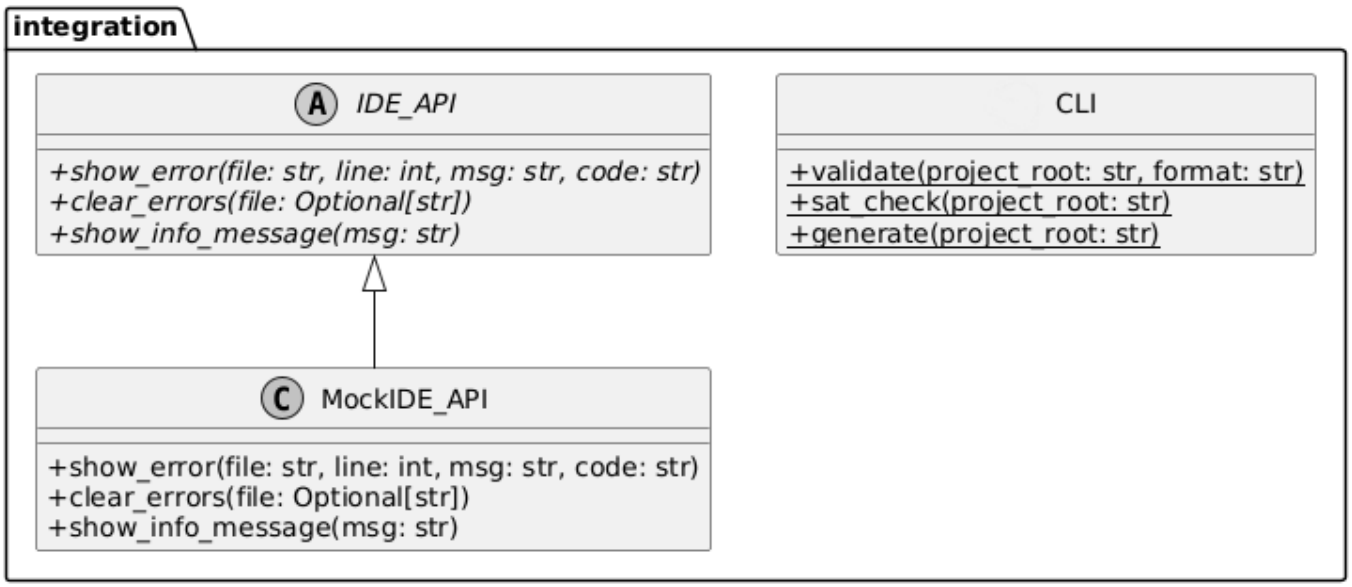

Рисунок 4.5 – Діаграма класів модуля integration

Робота плагіна являє собою послідовний конвеєр обробки даних, що проходить через описані вище архітектурні модулі. Цей процес складається з наступних кроків:

1. *Ініціація***.** Процес валідації запускається зовнішнім агентом – користувачем через командний рядок або IDE через програмний виклик (модуль integration).

2. *Парсинг***.** Модуль parsers отримує шлях до каталогу з вихідним кодом, аналізує файли Python та будує для них абстрактні синтаксичні дерева (AST).

3. *Побудова моделі***.** Спеціалізований ASTVisitor обходить дерева, видобуваючи інформацію про класи, методи та атрибути, і наповнює відповідні об'єкти внутрішньої моделі. Одночасно з цим модуль core будує в пам'яті граф залежностей G_{MB}, фіксуючи зв'язки між усіма елементами.

4. *Генерація артефактів***.** Модуль generators отримує готову внутрішню модель і серіалізує її у два вихідні файли: model.struct.json (структурне подання) та model.behav.xmi (поведінкове подання).

5. *Інкрементальна валідація***.** Модуль validators отримує інформацію про змінені файли (множина Δ). Використовуючи граф залежностей GMB, він визначає повну множину зачеплених елементів Affected(Δμ) і запускає алгоритм ValBeh для перевірки лише тих інваріантів, які стосуються цих елементів.

6. *Результат***.** За підсумками валідації система генерує структурований звіт про всі виявлені помилки та неузгодженості, який повертається користувачу через CLI у вигляді текстового виводу або через API у форматі JSON для подальшої обробки в IDE.

Таким чином, визначивши архітектуру плагіна та відповідальність його ключових модулів, наступним кроком є детальний розгляд програмної реалізації його найважливіших компонентів та механізмів.

4.2 Реалізація ключових компонентів плагіна

Фундаментом будь-якого програмного інструменту є його технологічний стек, вибір якого визначає не лише продуктивність та масштабованість, але й швидкість розробки та легкість супроводу. У випадку науково-дослідного прототипу, де пріоритетом є швидка ітерація та перевірка гіпотез, вибір стеку стає стратегічним рішенням, що відображає філософію самого дослідження. Тому обраний

технологічний стек є не випадковим набором інструментів, а продуманою системою, де кожен компонент обрано для максимальної ефективності у вирішенні специфічних завдань, що стоять перед плагіном.

В основі реалізації лежить мова програмування Python версії 3.12. Цей вибір обґрунтований її високорівневим синтаксисом, потужною стандартною бібліотекою та широкою екосистемою сторонніх пакетів, що сукупно дозволяє значно прискорити розробку наукомістких прототипів. Кросплатформеність мови гарантує, що розроблені інструменти можуть бути використані в різноманітних операційних середовищах без необхідності модифікації коду. Ключовими бібліотеками, що формують технологічне ядро плагіна, є:

− *ast.* Вбудований у стандартну бібліотеку Python модуль для роботи з абстрактними синтаксичними деревами. Його використання для синтаксичного аналізу вихідного коду Python є фундаментальним для реалізації модуля parsers. Вибір вбудованого модуля усуває потребу в зовнішніх залежностях для ключової функціональності парсингу. Це, в свою чергу, підвищує надійність та спрощує розгортання інструменту.

− *lxml*. Високопродуктивна бібліотека для обробки XML та HTML, що є Python-обгорткою над C-бібліотеками libxml2 та libxslt. Вона була обрана як основний інструмент для парсингу, маніпуляції та серіалізації XMI-документів. Алгоритми трансформації висувають критичні вимоги до XML-процесора: здатність ефективно обробляти потенційно великі XMI-файли та наявність потужного механізму для виконання складних запитів. Бібліотека lxml повністю задовольняє цим вимогам. Швидкість парсингу та обходу DOM-дерева в lxml є на порядки вищою, ніж у реалізацій на чистому Python, що забезпечує масштабованість прототипів. Ще більш важливою є повна та високопродуктивна реалізація мови запитів XPath 1.0, яка інтенсивно використовується для класифікації елементів моделі та пошуку специфічних атрибутів (xmi:idref) або тегів (JsonRef).

− *jsonschema*. Де-факто стандартна бібліотека для програмної валідації JSON-документів відносно схеми. Вона є основою для реалізації першого, найшвидшого рівня валідаційного конвеєра, забезпечуючи синтаксичну коректність структурного

подання моделі.

Можна зробити висновки, що такий вибір технологій є свідомим балансом між гнучкістю та формалізмом, як і сама модель з двома поданнями. Динамічна природа Python та його багата екосистема сприяють швидкому прототипуванню, що відповідає принципам Agile. Водночас, цілеспрямоване використання високопродуктивних бібліотек для критично важливих операцій та інструментів формальної валідації демонструє прихильність до строгості та масштабованості, що є характерним для парадигми MDE.

Як вже було сказано раніше в розділі 5, центральним архітектурним елементом плагіна, що забезпечує низьке зчеплення та високу гнучкість, є внутрішня модель. Це не просто структура даних, а механізм роз'єднання, що слугує канонічним, незалежним від формату представленням UML-моделі. Внутрішня модель є прямим фізичним втіленням формальної метамоделі.

Для програмного визначення метасутностей UML (наприклад, UMLClass, UMLOperation, UMLAttribute) використовуються стандартні dataclasses мови Python. Цей вибір обґрунтований їхньою лаконічністю, автоматичною генерацією стандартних методів (__init__, __repr__, __eq__) та підтримкою статичної типізації. Сам підхід підвищує читабельність та надійність коду, дозволяючи точно й компактно відобразити формальні визначення. Кожен dataclass є прямим аналогом метакласу UML, а його поля – аналогами атрибутів метакласу.

Практична реалізація орієнтованого графа залежностей є ключовим компонентом модуля core. Граф реалізовано у вигляді списків суміжності на основі вкладених словників Python. Ключами зовнішнього словника є унікальні ідентифікатори елементів моделі (вершини графа). Значенням для кожного ключа є словник, що класифікує вихідні ребра за трьома типами:

− 'structural': [...],

− 'semantic': [...],

− 'traceability': [...].

Як можна бачити, така структура даних є не лише ефективною з точки зору доступу до суміжних вершин, але й дозволяє алгоритму інкрементальної валідації

виконувати цільові, семантично обґрунтовані запити. Наприклад, для перевірки каскадних змін, пов'язаних з видаленням контейнера, аналізуються лише ребра типу structural, тоді як для перевірки міжподаннєвих інваріантів використовуються ребра traceability. Фактично, це є практичною реалізацією класифікації ребер, що є одним з наукових результатів роботи.

Модуль parsers виконує функцію «фронт-енду» системи, відповідаючи за перетворення синтаксичних конструкцій вихідного коду на семантично багаті об'єкти внутрішньої моделі. По суті, цей процес є актом формалізації, що перетворює неявне знання, закладене у вихідному коді, на явну, структуровану та верифіковану модель. Вихідний код Python містить імпліцитну структурну та семантичну інформацію. Наприклад, визначення класу class MyClass: *означає* існування метасутності UML Class з іменем MyClass; анотація типу attr: int *означає* існування атрибута з типом Integer. Модуль parsers робить це знання явним, підіймаючи семантику коду до рівня формальної моделі, що уможливлює всі подальші операції аналізу та валідації.

Процес аналізу починається зі зчитування вихідного файлу Python та виклику стандартної функції ast.parse(), яка повертає кореневий вузол ast.Module абстрактного синтаксичного дерева. AST є ієрархічним представленням граматичної структури коду, де кожен вузол відповідає певній синтаксичній конструкції.

Центральним елементом модуля є спеціалізований клас ASTParserVisitor, що успадковує стандартний клас ast.NodeVisitor. Цей клас реалізує патерн проектування «Відвідувач» (Visitor), що дозволяє відокремити логіку обробки вузлів від структури самого дерева. Для кожного типу вузла, який представляє інтерес для побудови моделі, в класі ASTParserVisitor реалізовано відповідний метод-відвідувач visit_<NodeType>().

Нижче описана реалізація ключових методів-відвідувачів.

visit_ClassDef(node). Цей метод викликається для кожного вузла типу ast.ClassDef, що представляє визначення класу. Логіка методу виконує такі кроки:

1. Вилучення імені класу з атрибута node.name.

2. Аналіз списку базових класів node.bases для фіксації відношень успадкування. Для кожного базового класу вилучається його ідентифікатор.

3. Створення нового екземпляра dataclass UMLClass та наповнення його отриманою інформацією.

4. Рекурсивний виклик self.generic_visit(node) для обробки тіла класу, що ініціює виклик методів visit_FunctionDef та visit_AnnAssign для всіх вкладених методів та атрибутів.

visit_FunctionDef(node). Цей метод обробляє визначення функцій (методів класу). Він вилучає ім'я методу (node.name), аналізує об'єкт node.args для отримання списку параметрів, їхніх імен та анотацій типів, а також вилучає анотацію типу значення, що повертається, з node.returns. На основі цих даних створюється та наповнюється екземпляр UMLOperation.

visit_AnnAssign(node). Цей метод є критично важливим для аналізу атрибутів класу з явними анотаціями типів. Він дозволяє надійно вилучити як ім'я атрибута, так і його тип. Ця інформація використовується для створення екземпляра UMLAttribute. Обробка саме цього типу вузла є необхідною для побудови повної та семантично коректної структурної моделі, оскільки вона дозволяє фіксувати не лише наявність атрибутів, але і їхні типи, що є ключовим для подальшої валідації.

Варто зазначити, що розроблена метамодель та алгоритми валідації є інваріантними до мови програмування. Прив'язка до Python стосується лише модуля парсингу . Для адаптації плагіна до інших мов достатньо реалізувати відповідний Visitor для AST цих мов, не змінюючи ядро системи та логіку валідації.

Отже, клас ASTParserVisitor рекурсивно обходить все синтаксичне дерево, ідентифікуючи ключові структурні елементи та послідовно будуючи в пам'яті об'єктне представлення UML-моделі, яке потім передається до модуля *core*.

Модуль generators виконує функцію «бек-енду» системи, відповідаючи за серіалізацію об'єктів внутрішньої моделі, що зберігаються в модулі core, у цільові файлові артефакти. Цей модуль є, по суті, «двомовним» серіалізатором, який транслює єдине джерело істини – внутрішню модель – на дві різні мови для двох різних аудиторій (розробників/систем контролю версій та CASE-засобів відповідно). Реалізація цього модуля є не простою операцією запису у файл, а включає вбудовані механізми контролю якості, що гарантують як внутрішню коректність кожного

артефакту, так і зовнішню консистентність між ними.

Генерація структурного подання (model.struct.json).

Процес генерації структурного подання реалізовано шляхом рекурсивного обходу об'єктного графа внутрішньої моделі та побудови відповідної ієрархічної структури зі словників та списків Python. Ця структура даних точно відображає ієрархію, визначену в уніфікованій JSON Schema з розділу 2.

Ключовим кроком, що забезпечує достовірність, є програмна валідація згенерованої структури даних *перед* її серіалізацією у файл. Для цього використовується бібліотека jsonschema та її функція validate(). Ця операція перевіряє відповідність згенерованих даних формальному контракту, визначеному у файлі schemas/structural_schema.json. Такий підхід реалізує принцип «коректність за конструкцією» (correctness by construction, CbC) на синтаксичному рівні. Він виступає як вбудований шлюз якості, що гарантує, що плагін ніколи не згенерує синтаксично невалідний JSON-артефакт, що є першим та найшвидшим рівнем у загальній стратегії валідації. Лише після успішного проходження валідації структура даних серіалізується у текстовий файл за допомогою стандартного модуля json.

Генерація поведінкового подання (model.behav.xmi).

Генерація XMI-документа є більш складним процесом, що реалізується за допомогою бібліотеки lxml. Об'єкти внутрішньої моделі, що відповідають поведінковим елементам (станам, переходам, повідомленням тощо), ітеративно обробляються для побудови DOM-дерева XMI-документа.

Центральним елементом цього процесу є фізична реалізація відношення відповідності μ. Для кожного поведінкового елемента у внутрішній моделі, який має семантичний зв'язок зі структурним елементом, виконується операція збагачення XMI-вузла. Програмно створюється вузол <xmi:Extension>, в який, відповідно до розробленого UML-профілю додається вузол <JsonRef>. Атрибуту jsonId цього вузла присвоюється унікальний ідентифікатор відповідного структурного елемента з внутрішньої моделі. Ця операція є фізичним втіленням абстрактного кортежу $(s,b) \in \mu$. Вбудовані таким чином теги JsonRef функціонують як «гіперпосилання» між двома згенерованими документами, дозволяючи валідатору та іншим компонентам системи

надійно відновлювати зв'язки між фізично розділеними, але логічно цілісними поданнями моделі.

Модуль validators є інтелектуальним ядром плагіна, що реалізує всі рівні перевірки консистентності. Його архітектура втілює принцип «fail-fast», створюючи багаторівневий валідаційний конвеєр. Кожен наступний рівень цього конвеєра має вищу обчислювальну вартість, але й забезпечує більшу семантичну глибину аналізу. Такий підхід є, по суті, системою управління обчислювальними ризиками: він оптимізує баланс між вартістю обчислень та ризиком пропустити архітектурний дефект. Прості, поширені помилки виявляються миттєво за допомогою «дешевих» перевірок, що запускаються часто. Більш складні семантичні неузгодженості виявляються швидко за допомогою інкрементального аналізу при кожній зміні. А глибокі, рідкісні, але критичні глобальні дефекти виявляються за допомогою «дорогих» формальних методів, що запускаються на вимогу або в рамках CI/CD. Все це робить застосування формальних методів практичним для робочих процесів.

Рівень 1: Синтаксична валідація. Цей рівень реалізовано, як було описано вище, безпосередньо в модулі generators під час генерації файлу model.struct.json. Використання бібліотеки jsonschema для перевірки відповідності згенерованих даних уніфікованій JSON Schema є найшвидшим рівнем, що відсіює базові структурні помилки, такі як відсутність обов'язкових полів, некоректний формат ідентифікаторів або дублювання елементів у масивах.

Рівень 2: Інкрементальна семантична валідація. Це ядро інтерактивної роботи плагіна, що запускається при кожній зміні вихідного коду. Воно реалізоване у компоненті ValidationEngine, який є програмним втіленням високоефективного інкрементального алгоритму ValBeh.

Операціоналізація алгоритму ValBeh відбувається наступним чином:

1. На вхід рушій отримує множину ідентифікаторів елементів Δ, які були безпосередньо змінені в результаті останньої операції редагування коду.

2. Рушій звертається до графа залежностей G_{MB}, що зберігається в модулі core, і виконує обхід графа (наприклад, в ширину) від вузлів з множини Δ. Результатом обходу є повна множина зачеплених елементів Affected(Δ), яка включає як самі

змінені елементи, так і всі елементи, що прямо чи опосередковано від них залежать.

3. Система ітерує *виключно* по елементах з множини Affected(Δ). Для кожного елемента, залежно від його типу (поля _type в dataclass), викликається відповідний набір функцій-валідаторів.

4. Кожен формальний інваріант (наприклад, SignatureMatches, LifelineCorrespondence, BehaviorExists) реалізовано як окрема, атомарна функція Python. Кожна така функція приймає на вхід один або кілька об'єктів внутрішньої моделі та повертає список виявлених порушень. Цей підхід є прямим практичним застосуванням теореми інкрементальної консистентності, яка математично гарантує, що успішне проходження локальної перевірки на множині Affected(Δ) є достатньою умовою для гарантування глобальної консистентності всієї моделі.

Рівень 3: Глибока формальна верифікація. Цей рівень призначений для виявлення складних транзитивних та глобальних властивостей моделі, які недоступні для локальних перевірок. Він реалізований у компоненті AlloyBridge.

Процес верифікації за допомогою AlloyBridge складається з наступних кроків:

1. Компонент отримує на вхід підмножину внутрішньої моделі, що підлягає аналізу (наприклад, всі класи та їхні відношення успадкування для перевірки ациклічності).

2. Програмно генерується текстовий файл специфікації мовою Alloy (.als). Наприклад, для перевірки ациклічності ієрархії успадкування генерується твердження assert NoCycles { no c: Class | c in c.^extends }, де ^extends є оператором транзитивного замикання в Alloy.

3. За допомогою стандартного модуля Python subprocess викликається зовнішній SAT-вирішувач Alloy (зазвичай розповсюджується як виконуваний .jar файл) як окремий дочірній процес.

4. Система очікує завершення процесу та аналізує його стандартний вивід (stdout). Якщо Alloy знаходить контрприклад (тобто екземпляр моделі, що порушує твердження), його текстовий вивід парситься для вилучення інформації про конкретні елементи, що утворюють дефект (наприклад, послідовність класів, що утворюють цикл успадкування). Ця інформація потім форматується у зрозуміле для користувача

повідомлення про помилку. Через високу обчислювальну складність SAT-аналізу, цей рівень валідації запускається лише за прямою командою користувача або в рамках автоматизованого конвеєра CI/CD.

При цьому треба зазначити, що для забезпечення повноцінного життєвого циклу моделі, окрім аналізу та валідації, необхідні механізми, що підтримують її еволюцію та сумісність із зовнішнім світом. Цю відповідальність несуть компоненти, що реалізують алгоритми двосторонньої синхронізації та трансформації.

Цей компонент є високорівневим оркестратором, який координує роботу раніше описаних модулів для підтримки моделі в консистентному стані. Його робочий цикл виглядає наступним чином:

1. Процес ініціюється зовнішньою подією, наприклад, зміною у файловій системі, що відстежується плагіном в IDE.

2. Викликається модуль parsers для оновлення внутрішньої моделі відповідно до нового стану вихідного коду.

3. Спеціалізована логіка синхронізації аналізує відмінності між старим та новим станом внутрішньої моделі та, за необхідності, вносить додаткові автоматичні модифікації для відновлення узгодженості (наприклад, створює «заглушку» для нової операції, щоб задовольнити інваріант BehaviorExists).

4. Викликається ValidationEngine для запуску інкрементальної валідації на множині зачеплених елементів.

5. Якщо валідація успішна, викликається модуль generators для серіалізації оновленої та узгодженої внутрішньої моделі назад у цільові файли model.struct.json та model.behav.xmi.

Для забезпечення сумісності з традиційними CASE-засобами, що оперують монолітними XMI-моделями, реалізовано два окремі скрипти, що втілюють алгоритми трансформації.

Перший – це пряма трансформація (XMI → JSON + XMI). Цей скрипт використовує бібліотеку lxml та комплексні XPath-запити (наприклад, //uml:Class | //uml:Package |...) для ефективної класифікації всіх елементів вхідного XMI-файлу на множини StructSet та BehSet. Потім він послідовно виконує операції генерації JSON

та редукції/збагачення XMI, як це детально описано в алгоритмі.

Другий скрипт – це зворотна трансформація (JSON + XMI → XMI). Цей скрипт одночасно парсить обидва вхідні файли. Він ітерує по JSON-структурі, програмно генеруючи відповідні XMI-фрагменти (<uml:Class>, <uml:Property> тощо) за допомогою lxml. Потім ці фрагменти вставляються у відповідні місця DOM-дерева, завантаженого з поведінкового XMI-файлу. На завершальному етапі виконується критична операція розв'язання посилань: скрипт знаходить усі трасувальні теги JsonRef і замінює їх на стандартні посилання xmi:idref, відновлюючи цілісність монолітної моделі.

Ці скрипти є не лише технічними утилітами, а й слугують практичним доказом семантичної безвтратності та повноти запропонованого підходу.

Таблиця 4.1 – Поєднання теоретичних конструктів із програмними компонентами

Теоретичний конструкт	Програмний компонент реалізації
Метамодель $M=(M_S,M_B,\mu)$	Python dataclasses в модулі core, що визначають структуру UMLClass, UMLOperation та інших.
Уніфікована JSON Schema	Файл schemas/structural_schema.json та його використання в generators через бібліотеку jsonschema.
Спеціалізований UML-профіль	Логіка генерації вузлів <xmi:Extension> та <JsonRef> у модулі generators/xmi_generator.py.
Граф залежностей G_{MB}	Структура даних на основі словників у core/dependency_graph.py, що реалізує списки суміжності.
Алгоритм інкрементальної валідації ValBeh	Компонент ValidationEngine у модулі validators/validation_engine.py.
Теорема інкрементальної консистентності	Логіка локалізованої перевірки лише для множини Affected в ValidationEngine.
Алгоритми трансформації (RTE)	Окремі скрипти scripts/forward_transform.py та scripts/backward_transform.py.

Детальний опис програмної реалізації ключових компонентів плагіна підтверджує, що розроблений у дисертації науково-методичний апарат є не лише

теоретично коректним, але й практично реалізовним. Продуманий вибір технологічного стеку, модульна архітектура, що базується на принципі розділення відповідальності, та точне втілення формальних методів і алгоритмів у програмному коді створюють надійну основу для інструментального засобу, здатного ефективно вирішувати поставлену задачу забезпечення консистентності UML-моделей в умовах сучасної ітеративної розробки. Таблиця 4.1 наочно демонструє прямий зв'язок між кожним теоретичним конструктом роботи та його конкретним, функціонуючим програмним аналогом, слугуючи матрицею трасування для всієї дисертації.

4.3 Інтеграція плагіна в процеси розробки

Розглянувши в попередніх підрозділах внутрішню архітектуру та програмну реалізацію ключових компонентів плагіна, необхідно перейти до аналізу способів взаємодії розробника з системою та її інтеграції в існуючі інженерні практики. Практична цінність будь-якого інструментального засобу в програмній інженерії визначається не лише його внутрішньою обчислювальною потужністю, але й тим, наскільки органічно та з мінімальними накладними витратами він може бути вбудований у щоденні робочі процеси.

Для вирішення цієї задачі архітектура плагіна передбачає вже описаний спеціалізований модуль integration, який слугує зовнішнім фасадом системи та має інтерфейси CLI та API, які орієнтовані як на CI/CD, так і на розробника в рамках IDE. Тобто такий двокомпонентний підхід є не просто набором функцій, а свідомою архітектурною стратегією. Інтеграція в IDE підтримує «внутрішній цикл» розробки, допомагаючи інженеру уникати помилок на етапі їх виникнення. CLI, своєю чергою, підтримує «зовнішній цикл», виступаючи як фінальний, автоматизований шлюз якості на рівні команди та репозиторію. Разом вони створюють комплексну систему управління архітектурною цілісністю, що є одночасно і превентивною, і директивною, демонструючи зріле розуміння потреб сучасної інженерії програмного забезпечення.

Інтерфейс командного рядка розроблено спеціально для забезпечення

можливості автоматизації та вбудовування перевірки консистентності в автоматизовані конвеєри збірки та тестування. Тому це перетворює валідацію моделі з опціональної ручної дії на обов'язковий, невід'ємний етап процесу контролю якості, що дозволяє командам розробки впроваджувати та підтримувати сурові архітектурні політики.

Основні команди CLI зведено в табл. 4.2.

Таблиця 4.2 – Основні команди CLI

Команда	Призначення	Приклад використання
uml-consistency generate	Запускає повний аналіз (парсинг) вихідного коду у вказаному каталозі та генерує проміжні артефакти моделі (model.struct.json та model.behav.xmi) у службовій директорії .uml-consistency/.	poetry run uml-consistency generate --project-root./project
uml-consistency validate	Запускає повний аналіз (парсинг) вихідного коду у вказаному каталозі та генерує проміжні артефакти моделі (model.struct.json та model.behav.xmi) у службовій директорії .uml-consistency/.	poetry run uml-consistency validate --format json --project-root./project
uml-consistency sat-check	Запускає найбільш ресурсоємну глибоку верифікацію (рівень 3) за допомогою SAT-вирішувача для перевірки складних глобальних інваріантів (напр., ациклічність).	poetry run uml-consistency sat-check --project-root./project

Структура команд CLI є практичним втіленням багаторівневої стратегії валідації, описаної раніше. Вона дозволяє ефективно управляти обчислювальними витратами, що є ключовим для подолання одного з головних бар'єрів на шляху до

впровадження формальних методів у промислову розробку – їхньої високої обчислювальної вартості. Команда validate, що виконує швидкі перевірки рівнів 1 та 2, є достатньо легкою для запуску при кожному коміті, забезпечуючи швидкий зворотний зв'язок щодо більшості поширених помилок. Натомість, команда sat-check, що запускає більш дорогий з точки зору часу та ресурсів аналіз рівня 3, тому призначена для більш рідкісного використання, наприклад, як обов'язковий етап перед злиттям великої функціональної гілки в основну або в рамках нічних збірок. Такий підхід робить застосування глибокої формальної верифікації прагматичним та економічно доцільним.

Практична інтеграція в CI/CD-конвеєр демонструється на прикладі конфігураційного файлу для GitHub Actions. Наведений нижче фрагмент коду (.github/workflows/consistency-check.yml) ілюструє, як на подію pull_request до гілки main автоматично запускається крок валідації.

```
name: UML Model Consistency Check

on:
  pull_request:
    branches: [ main ]

jobs:
  validate-model:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.12'
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install poetry
        poetry install
    - name: Run UML Consistency Validation
      id: validation
      run: |
```

```
              poetry run uml-consistency validate --format json --project-root./src >
validation_results.json
              echo "exit_code=$?" >> $GITHUB_ENV
          - name: Check validation results
            if: env.exit_code!= 0
            run: |
              echo "Model consistency check failed. See details below:"
              cat validation_results.json
              exit 1
```

У цьому сценарії команда uml-consistency validate запускається з опцією --format json, що дозволяє зберегти результати у машиночитному форматі. Наступний крок аналізує код повернення (exit code) процесу. Якщо валідація виявляє помилки, команда завершується з ненульовим кодом, що призводить до провалу всієї роботи в GitHub Actions. Це, своєю чергою, автоматично блокує можливість злиття запиту на злиття, ефективно запобігаючи потраплянню архітектурних дефектів до основної кодової бази.

Якщо CLI забезпечує автоматизований контроль якості на рівні команди, то інтеграція з IDE є ключовою перевагою для окремого розробника. Її мета – це не просто виявляти помилки постфактум, а робити швидко, в контексті та з мінімальним когнітивним навантаженням, реалізуючи принцип «shift-left» для архітектурної якості. Цей прицип перетворює архітектурну валідацію з дорогої, реактивної діяльності на дешеву, проактивну та превентивну практику, що має значний позитивний економічний ефект, оскільки вартість виправлення дефекту на етапі його створення є на порядки нижчою, ніж на етапах тестування чи експлуатації.

Для демонстрації цього механізму розглянемо наступний контрольний приклад.

Сценарій: розробник виконує типову операцію рефакторингу у вихідному коді на мові Python, перейменовуючи метод device.toggle_status() на device.set_active().

Помилка: розробник забуває оновити виклик цього методу в docstring-коментарі, який містить текстовий опис діаграми послідовності за допомогою спеціального тегу: @sequence... -> device : toggle_status().

У традиційному процесі розробки така помилка, будучи семантичною, а не

синтаксичною, залишилася б непоміченою компілятором та стандартними лінтерами. Вона могла б проявитися значно пізніше, наприклад, під час рев'ю коду або, що гірше, призвести до розсинхронізації між кодом та архітектурною документацією. Розроблений плагін, натомість, реагує миттєво.

Нижче наведено покроковий аналіз процесів, що відбуваються «під капотом» системи.

Крок 1. Інкрементальний аналіз при збереженні. У момент збереження файлу в IDE, плагін через свій API ініціює аналіз. Замість повної перевірки всього проекту, запускається високоефективний інкрементальний парсер. Спеціалізований клас ASTParserVisitor обходить AST зміненого файлу та швидко ідентифікує точну зміну, а саме – це перейменування вузла ast.FunctionDef з toggle_status на set_active.

Крок 2. Аналіз впливу за допомогою графа залежностей. Система звертається до графа залежностей G_{MB}, що зберігається в пам'яті. Рухаючись по ребрах графа, система миттєво визначає повну множину зачеплених елементів Affected(Δ). У даному випадку, ключову роль відіграє трасувальне ребро, яке є фізичною реалізацією формального відношення відповідності μ. Це ребро пов'язує змінений об'єкт UMLOperation (відповідний методу set_active) з об'єктом UMLMessage у поведінковому поданні, який був згенерований з docstring-коментаря.

Крок 3. Локалізована семантична валідація. Запускається валідаційний рушій ValidationEngine, що реалізує алгоритм ValBeh. Важливо підкреслити, що перевірка виконується лише для елементів з множини Affected(Δ). Валідатор перевіряє інваріант SignatureMatches (або більш базовий AnchorExists) для об'єкта UMLMessage. Оскільки операції toggle_status() більше не існує в класі Device, інваріант порушується.

Крок 4. Швидкий та контекстний зворотний зв'язок. Плагін, отримавши результат валідації, негайно передає його розробнику через API інтегрованого середовища розробки. Розробник бачить, що рядок коду @sequence... -> device : toggle_status() підсвічується як помилка безпосередньо в редакторі. При наведенні курсора на підсвічену ділянку з'являється інформативне повідомлення, наприклад: «Помилка консистентності: Повідомлення посилається на неіснуючу операцію

'toggle_status' у класі 'Device'. Можливо, ви мали на увазі 'set_active'?».

В результаті, розробник отримує точну, контекстну та дієву інформацію про архітектурний дефект у момент його створення, що дозволяє виправити помилку за лічені секунди, підтримуючи модель та код у постійно узгодженому стані.

4.4 Експериментальне дослідження та порівняння з аналогами

Завершальним етапом валідації є проведення строгого експериментального дослідження. Метою даного дослідження є кількісна оцінка запропонованого інкрементального підходу та порівняння його ключових показників з існуючими промисловими аналогами. В попередніх підрозділах вже було показано, що розроблене рішення є теоретично обґрунтованим. Практичні ж результати дозволять перейти від концептуальних доказів до вимірюваних фактів, які остаточно покажуть, чи являється новий підхід практично ефективнішим за існуючі альтернативи, вирішуючи при цьому фундаментальну проблему поєднання формалізму та гнучкості.

4.4.1 Оцінка ефективності

Для забезпечення наукової валідності, відтворюваності та статистичної значущості отриманих результатів було розроблено сувору методику експериментального дослідження. Мета експерименту полягає не лише у фіксації абсолютних показників швидкодії, а й у дослідженні динаміки поведінки алгоритмів при зміні масштабів вхідних даних, що дозволяє підтвердити або спростувати теоретичні оцінки масштабованості.

Усі тестові випробування проводилися на уніфікованій апаратній платформі, конфігурація якої є репрезентативною для типового високопродуктивного робочого місця розробника програмного забезпечення або виділеного вузла CI/CD конвеєра. Вибір такої конфігурації дозволяє мінімізувати вплив апаратних обмежень на

результати вимірювань та екстраполювати отримані дані на реальні індустріальні умови експлуатації.

– *Центральний процесор*. Intel Core i7 (архітектура x86-64, тактова частота 3.6 ГГц, 8 фізичних ядер, підтримка багатопотоковості). Висока тактова частота є критичною для однопотокових операцій парсингу, характерних для багатьох CASE-засобів.

– *Оперативна пам'ять*. 32 ГБ DDR4. Обсяг пам'яті обрано з запасом для виключення впливу свопінгу на результати вимірювання часу виконання, що дозволяє оцінювати «чисту» ефективність алгоритмів управління пам'яттю.

– *Дискова підсистема*. SSD NVMe. Використання швидкісного накопичувача є принциповим для мінімізації латентності файлових операцій вводу-виводу, оскільки процес валідації передбачає інтенсивне читання та запис артефактів моделі (JSON та XMI файлів).

– *Операційне середовище*. 64-бітна версія Windows 10 Home з відключеними фоновими процесами оновлення та індексації для забезпечення стабільності вимірювань.

В якості вхідних даних для експериментів використовувалися два типи наборів моделей, що дозволяють оцінити як поведінку системи в типових умовах, так і її граничні можливості. Перший – це реальний типовий проект середнього розміру, що складається з 50 класів та близько 500 методів, реалізований мовою Python. Цей набір даних використовується для базового порівняння в рамках етапу 1 і відображає структуру залежностей, характерну для стандартних архітектурних патернів, де середній рівень зв'язності є помірним. Другий – це синтетичні набори даних, серія автоматично згенерованих моделей зі строго контрольованими та варіативними параметрами розмірності (N) та щільності зв'язків (d). Ці набори використовуються для стрес-тестування та перевірки асимптотичної поведінки алгоритмів (етапи 2 та 3). Генерація виконувалася спеціалізованим скриптом, що створює валідні XMI та JSON структури, забезпечуючи діапазон масштабування від 50 до 5000 класів, що покриває спектр від малих бібліотек до великих корпоративних систем.

Для проведення порівняльного аналізу було обрано два найбільш поширені в індустрії CASE-засоби, які представляють різні, іноді протилежні підходи до архітектури моделювання, що дозволяє позиціонувати розроблений плагін у системі координат «гнучкість – строгість»:

– *Плагін*. Реалізує запропонований метод інкрементальної валідації на основі графа залежностей G_{MB} та метамоделі з двома поданнями. Він використовує оптимізовані хеш-індекси для швидкого пошуку елементів та алгоритм локалізованого обходу графа для визначення області впливу змін.

– *StarUML* (версія 6.3). Популярний легковаговий інструмент моделювання, що використовує власний JSON-орієнтований формат зберігання (.mdj). Він обраний як представник інструментів, орієнтованих на гнучкість та швидкість інтерфейсу, але таких, що архітектурно покладаються на повну ревалідацію моделі при перевірці правил, оскільки не підтримують гранулярного відстеження семантичних залежностей у реальному часі.

– *Visual Paradigm* (версія 17.3). Комплексне середовище моделювання корпоративного рівня («важковаговий» інструмент). Обраний як еталон функціональної повноти та суворої відповідності стандартам OMG. Однак, його архітектура, побудована на базі Java/JVM та складних внутрішніх метамоделей, характеризується високою ресурсоємністю, що робить його ідеальним опонентом для перевірки гіпотези про переваги легковагових рішень.

Експериментальне дослідження було структуровано у три послідовні етапи, кожен з яких фокусувався на ізоляції та перевірці окремого фактора продуктивності: базове порівняння на типових задачах, аналіз чутливості до масштабу та дослідження впливу топології залежностей.

Етап 1. Базове порівняння на типовому проекті

На першому етапі дослідження було змодельовано найбільш поширений та критичний сценарій повсякденної роботи розробника: внесення локальної атомарної зміни в архітектуру проекту під час ітеративної розробки. Цей сценарій є визначальним для оцінки придатності інструменту до використання в режимі активної валідації безпосередньо в IDE. Психоергономічні дослідження показують,

що затримка реакції системи понад 0.1–1.0 секунду призводить до втрати стану потоку та зниження когнітивної продуктивності інженера, тому здатність інструменту вкладатися у цей часовий ліміт є критичною вимогою.

Сценарій експерименту. У вихідному коді реального проекту (50 класів, 500 методів) було виконано операцію рефакторингу: перейменування одного публічного методу в ключовому класі бізнес-логіки. Ця зміна є семантично значущою, оскільки потенційно впливає на консистентність ряду поведінкових діаграм (наприклад, діаграм послідовності, де цей метод викликається як повідомлення, або діаграм станів, де він виступає тригером). Для кожного інструменту вимірювався час, що проходить від моменту ініціації валідації (наприклад, подія збереження файлу або виклик команди меню) до отримання повного звіту про статус консистентності, а також пікове споживання оперативної пам'яті під час цього процесу. Вимірювання проводилися 10 разів для кожного інструменту з подальшим усередненням результатів для нівелювання випадкових флуктуацій операційної системи.

Результати вимірювань. Усереднені результати порівняння наведено в табл. 4.3.

Таблиця 4.3 – Порівняння продуктивності валідації на типовому проекті

Інструмент	Метод валідації	Час виконання (сек)	Споживання RAM (МБ)
Плагін	Повна	3.8	226
Плагін	Інкрементальна	0.3	110
StarUML	Повна	4.4	248
Visual Paradigm	Повна	6.8	274

Аналіз результатів. Отримані емпіричні дані демонструють кардинальну, якісну перевагу запропонованого методу навіть на проектах невеликого та середнього розміру, які вважаються легковагими для сучасних комп'ютерів.

У режимі повної валідації, коли плагін, подібно до промислових аналогів, виконує повний аналіз усієї моделі, час виконання становить 3.8 с, а споживання пам'яті – 226 МБ. Ці показники є порівнянними з результатами StarUML (4.4 с, 248

МБ) та дещо кращими за Visual Paradigm (6.8 с, 274 МБ). Така різниця пояснюється відсутністю у плагіна накладних витрат на графічний інтерфейс користувача та оптимізованим стеком технологій (використання легковагових бібліотек замість важких фреймворків Electron або Java Swing). Це підтверджує, що сам по собі валідаційний рушій плагіна є ефективним, проте не демонструє надприродної швидкодії при виконанні класичного повного перебору O(N).

Кардинальна ж перевага розкривається при використанні методу інкрементальної валідації. Час реакції скорочується до 0.3 с, що у 14.6 разів швидше за StarUML та у 22.6 разів швидше за Visual Paradigm. Такий стрибок продуктивності пояснюється фундаментальною алгоритмічною відмінністю: тоді як аналоги змушені щоразу виконувати повний розбір та перевірку всіх елементів (оскільки їхня архітектура жорстко пов'язана з монолітним станом моделі), розроблений плагін використовує заздалегідь побудований граф залежностей. Це дозволяє локалізувати перевірку виключно на множині зачеплених елементів, яка в даному сценарії містить лише змінений метод та декілька пов'язаних повідомлень.

Також в інкрементальному режимі спостерігається більш ніж двократна економія оперативної пам’яті (110 МБ проти 248–274 МБ). Це досягається завдяки тому, що система не потребує одночасного утримання в пам'яті повних об'єктних структур для всієї моделі, підвантажуючи або оновлюючи лише необхідні фрагменти JSON/XMI структур. Низьке споживання пам'яті є критично важливим для сценаріїв хмарного розгортання в контейнерах (наприклад, Docker-агенти в CI/CD), де ресурси часто лімітовані.

Отже, результати цього етапу підтверджують: хоча в режимі повної перевірки плагін демонструє очікувану базову ефективність на рівні аналогів, саме застосування інкрементального підходу забезпечує той рівень реактивності (<1 с), який є необхідним для інтеграції валідації безпосередньо в «гарячий» цикл розробки.

Етап 2. Дослідження масштабованості

Критичним викликом для будь-якого методу аналізу моделей у контексті MDE є його здатність ефективно працювати з великими промисловими системами. Традиційні алгоритми валідації, що реалізуються в більшості CASE-засобів, мають

складність O(N), де N – сумарна кількість елементів у моделі. Це створює проблему лінійної деградації продуктивності: зі зростанням масштабу проекту час перевірки зростає пропорційно (або навіть поліноміально у випадку перехресних перевірок складної логіки), що неминуче перетворює валідацію на вузьке місце у швидкісних CI/CD конвеєрах. Метою другого етапу є емпірична перевірка теоретичного твердження, виведеного в розділі 3, про те, що час роботи запропонованого інкрементального методу слабко залежить від загального розміру моделі N, а визначається переважно локальними характеристиками внесеної зміни.

Сценарій експерименту. Для перевірки гіпотези було згенеровано ряд синтетичних моделей зростаючої розмірності: 100, 500, 1000 та 5000 класів. У кожній моделі зберігалася фіксована середня топологія (середня щільність зв'язків), що імітує нормальний розподіл залежностей у добре структурованому ПЗ (кожен клас має асоціації в середньому з 3-5 іншими класами). У кожному експерименті вносилася однакова за складністю атомарна зміна (зміна атрибута одного класу), що дозволяє ізолювати фактор розміру моделі N як єдину змінну.

Результати вимірювань. Динаміка зростання часу виконання валідації залежно від розміру моделі наведена в табл. 4.4.

Таблиця 4.4 – Залежність часу валідації від розміру моделі при фіксованій зміні

Кількість класів (N)	Плагін (сек)	StarUML (сек)	Visual Paradigm (сек)
50	0.30	4.4	6.8
100	0.35	8.2	11.5
500	0.40	35.0	42.0
1000	0.50	68.0	85.0
5000	0.80	310.0	380.0

Аналіз результатів. Інтерпретація даних таблиці 4.4 дозволяє виявити відмінності між порівнюваними підходами, яка розширюється зі зростанням масштабу системи.

Інструменти StarUML та Visual Paradigm демонструють чітку, майже ідеальну лінійну залежність часу виконання від розміру проекту. При зростанні моделі до 5000

класів час валідації досягає катастрофічних значень – понад 5 хвилин (310 с) для StarUML та понад 6 хвилин (380 с) для Visual Paradigm. У реальному виробничому процесі така затримка є блокуючим фактором. Розробник не може чекати 6 хвилин після кожного збереження файлу для перевірки архітектурної цілісності. Це призводить до того, що в реальних проектах автоматичну валідацію часто відключають або переносять на нічні збірки, що відкриває вікно можливостей для накопичення архітектурного дрейфу та помилок, виправлення яких на наступний день коштує значно дорожче.

Час виконання валідації розробленим плагіном демонструє поведінку, близьку до O(1) відносно N. При збільшенні розміру моделі у 100 разів (з 50 до 5000 класів), час перевірки зріс лише у 2.6 рази (з 0.3 до 0.8 с), залишаючись у межах однієї секунди. Це підтверджує теоретичну оцінку складності O(k·d). Оскільки кількість змін k залишається постійною (k=1), а локальна щільність зв'язків d у синтетичних моделях не змінюється із зростанням N, основний обсяг обчислювальної роботи залишається стабільним. Незначне абсолютне зростання часу (від 0.3 до 0.8 с) пояснюється логарифмічним зростанням накладних витрат на пошук елементів у хеш-таблицях великого розміру та первинним завантаженням індексів графа залежностей у пам'ять.

Цей результат є ключовим доказом того, що запропонований метод забезпечує фактично необмежену масштабованість процесу валідації. Він дозволяє застосовувати суворі методи забезпечення консистентності навіть для надвеликих систем (монолітів або розподілених систем з тисячами компонентів) без деградації продуктивності інструментарію, усуваючи залежність швидкості роботи інженера від загального обсягу кодової бази.

Етап 3. Вплив щільності зв'язків

Якщо попередній етап досліджував залежність від глобального параметра N, то цей етап присвячено глибинному аналізу впливу параметра d, який фігурує у теоретичній формулі складності як мультиплікатор. Метою є перевірка поведінки алгоритму у найгіршому випадку та дослідження чутливості методу до якості архітектури.

Сценарій експерименту. На фіксованому розмірі моделі (N=500 класів), що відповідає точці середнього навантаження, було змодельовано дві архітектурні ситуації, що представляють різні топології графа залежностей:

– *Sparse (слабкий зв'язок)*. Зміна вноситься у периферійний клас "Leaf Node", що має мінімальну кількість залежностей (d ≈ 2). Це відповідає зміні в ізольованому модулі або утилітарному класі.

– *Dense (сильний зв'язок).* Зміна вноситься у центральний, сильно зв'язаний клас "God Object" або "Core Service", від якого безпосередньо залежать десятки інших компонентів(d ≈ 50). Це імітує зміну в ядрі системи, яка має потенціал каскадного порушення консистентності.

Результати вимірювань. Порівняльні дані наведено в табл. 4.5.

Таблиця 4.5 – Вплив щільності залежностей на час валідації (N=500)

Сценарій	Щільність зв'язків (d)	Плагін (сек)	StarUML (сек)	Visual Paradigm (сек)
Sparse	Низька (d ≈ 2)	0.15	~35.0	~42.0
Dense	Висока (d ≈ 50)	0.65	~35.0	~42.0

Аналіз результатів. Дані цього етапу розкривають адаптивну природу запропонованого алгоритму на відміну від статичної природи традиційних підходів.

Час роботи StarUML та Visual Paradigm залишається статистично незмінним (~35 с та ~42 с) незалежно від того, який елемент змінюється. Це підтверджує, що їхні алгоритми виконують повне сканування моделі, не враховуючи контекст та семантику зміни. Вони витрачають значні обчислювальні ресурси на перевірку як ізольованих змін, що не несуть ризику, так і критично важливих змін. Це свідчить про неефективність використання ресурсів процесора.

В той же час розроблений плагін демонструє пряму кореляцію часу виконання зі щільністю зв'язків, що є ознакою інтелектуальної поведінки алгоритму. У сценарії *Sparse* час скорочується до рекордних 0.15 с (порівняно з 0.40 с середнього часу на етапі 2). Граф залежностей дозволяє миттєво визначити, що зміна локалізована, і відсікти іншу частину моделі від перевірки. У сценарії *Dense* час зростає до 0.65 с. Це

зростання є очікуваним і бажаним, оскільки воно відображає реальну необхідність перевірки 50 залежних елементів для гарантування цілісності. Алгоритм автоматично виділяє більше ресурсів там, де ризик порушення консистентності вищий.

Різниця у часі виконання між сценаріями Sparse та Dense (приблизно у 4 рази) корелює зі зростанням кількості залежностей, що потребують аналізу. Важливо підкреслити, що навіть у теоретично найгіршому випадку за рахунок інкрементальних оновлень плагін працює у значно швидше за аналогічні інструменти. Це свідчить про колосальний запас продуктивності методу. Навіть при роботі з погано спроектованими, сильно зв'язаними архітектурами, запропонований інструмент залишається придатним для інтерактивного використання, забезпечуючи швидкість, недосяжну для традиційних засобів.

Слід зазначити, що порівняння проводилося між CLI-інструментом та GUI-середовищами. Виграш у часі обумовлений не лише алгоритмічною ефективністю, але й відсутністю накладних витрат на графічний інтерфейс, що і є цільовим сценарієм для CI/CD.

Таким чином, експериментальна валідація підтверджує високу ефективність, масштабованість та практичну цінність розробленого науково-методичного апарату, доводячи його перевагу над існуючими підходами у контексті сучасних інкрементальних процесів розробки програмного забезпечення.

4.4.2 Оцінка достовірності

Цей підрозділ представляє повний, відтворюваний та методологічно обґрунтований експеримент, спрямований на об'єктивну оцінку достовірності розробленого плагіна у порівнянні з провідними промисловими CASE-засобами. Експеримент розроблено відповідно до усталених принципів емпіричної інженерії програмного забезпечення для забезпечення валідності, надійності та можливості узагальнення отриманих висновків.

Ключовий показник дослідження, *достовірність*, являє собою здатність інструменту коректно ідентифікувати семантичні дефекти. Для забезпечення

об’єктивної та кількісної оцінки якості виявлення дефектів було застосовано стандартні метрики з галузі інформаційного пошуку та машинного навчання, що дозволяють оцінити як здатність інструменту знаходити існуючі помилки, так і його схильність до хибних спрацювань.

Повнота (Recall). Ця метрика вимірює здатність інструменту знаходити всі релевантні дефекти. Вона розраховується як відношення кількості правильно виявлених дефектів (True Positives, TP) до загальної кількості дефектів у моделі, що є сумою правильно виявлених TP та пропущених дефектів (False Negatives, FN). Висока повнота є критично важливою для систем, де пропуск помилки може мати серйозні наслідки. Розраховується за формулою:

$$Recall = \frac{TP}{TP + FN}$$

Точність (Precision). Ця метрика показує, яка частка з усіх повідомлень інструменту про помилки є дійсно коректними. Вона розраховується як відношення кількості правильно виявлених дефектів TP до загальної кількості повідомлень, згенерованих інструментом, що є сумою правильно виявлених дефектів TP та хибних спрацювань FP. Висока точність означає, що інструмент не генерує хибних сповіщень, що є важливим для підтримки довіри розробника. Розраховується за формулою:

$$Precision = \frac{TP}{TP + FP}$$

F1-міра (F1-score). Є гармонійним середнім між точністю та повнотою, забезпечуючи єдиний, збалансований показник загальної точності інструменту. F1-міру обрано як основну метрику для перевірки гіпотез, оскільки вона штрафує системи, що мають значний перекіс в бік або високої повноти, або високої точності за рахунок іншої метрики. Розраховується за формулою:

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Наступним етапом необхідно провести створення об'єктивного тестового набору даних, який буде гарантувати, що всі інструменти оцінюються в абсолютно однакових умовах, усуваючи будь-які переваги, пов'язані зі знанням пропрієтарних форматів або особливостей внутрішньої реалізації.

Цей процес складається з двох послідовних кроків: формування моделі та внесення в неї контрольних дефектів в кількості 35 штук за допомогою бібліотеки lxml для точних маніпуляцій з XML-структурою мовою Python. В якості контрольного проекту планується відтворити тестовий проект із попереднього проекту (50 класів, 500 методів). А самі контрольні дефекти можна розділити на наступні категорії.

– *Невідповідність сигнатур операцій* (10 шт.). Скрипт знаходить пару елементів <message> та відповідний йому <ownedOperation>. Потім він змінює атрибут type одного з <ownedParameter> в операції, не змінюючи відповідний аргумент у повідомленні.

– *Висячі посилання* (8 шт.): Скрипт знаходить пару <lifeline> та відповідний йому <packagedElement xmi:type="uml:Class">. Потім він модифікує значення атрибута xmi:id елемента класу, свідомо порушуючи посилання в атрибуті represents елемента <lifeline>, створюючи класичне висяче посилання.

– *Використання застарілих типів даних* (7 шт.). Скрипт ідентифікує пов'язану пару елементів, наприклад, атрибут класу та InputPin у діаграмі діяльності, що його використовує. Він змінює тип атрибута класу, залишаючи тип InputPin без змін.

– *Порушення інкапсуляції* (10 шт.). Скрипт знаходить елемент <ownedOperation> з атрибутом visibility="private". Потім він створює або модифікує елемент <message>, що походить від лінії життя, яка представляє інший клас, і націлює його на цю приватну операцію, що є прямим порушенням принципів інкапсуляції.

Для всебічної оцінки надійності інструментів було розроблено двохетапний

протокол експерименту. Такий дизайн дозволяє не лише порівняти інструменти між собою, але й ізолювати та виміряти вплив інтероперабельності даних на ефективність їхніх валідаційних механізмів.

Етап 1. Валідація в рідних середовищах. Мета цього експерименту – встановити максимальну потенційну ефективність кожного інструменту, оцінюючи його в оптимізованому, пропрієтарному середовищі. Результати цього етапу слугують найкращим сценарієм, відносно якої буде вимірюватися деградація продуктивності на другому етапі.

Процедура виконується наступним чином.

1. Для StarUML та Visual Paradigm базова UML-модель була відтворена вручну елемент за елементом безпосередньо в середовищі кожного інструменту, що забезпечило створення нативних проектів у форматах .mdj та .vpp відповідно. Для розробленого плагіна використовувалася його власна згенерована модель.

2. Далі, 35 семантичних дефектів були послідовно внесені в кожну нативну модель. Процес внесення дефектів суворо слідував заздалегідь підготовленому скрипту, щоб гарантувати, що семантичний зміст кожного дефекту є ідентичним його XMI-аналогу.

3. Для кожного інструменту була виконана вбудована команда валідації моделі.

4. Усі згенеровані повідомлення про помилки та попередження були експортовані та збережені у текстовий файл для подальшого аналізу.

Етап 2. Валідація в умовах інтероперабельності. Мета цього етапу – оцінити достовірність кожного інструменту в більш реалістичному сценарії, де він змушений обробляти та аналізувати стандартизований зовнішній артефакт у форматі XMI. Цей етап є стрес-тестом для механізмів імпорту та валідації зовнішніх даних.

Процедура виконується наступним чином.

1. Плагін формує початкові проміжні артефакти: структурне та поведінкове подання.

2. Механізм зворотної трансформації плагіна об'єднує ці два подання в єдиний, монолітний та повністю консистентний файл, що відповідає стандарту UML/XMI 2.5, після чого в нього вносяться 35 контрольованих дефектів.

3. Перед кожним запуском для кожного інструменту було виконано очищення кешів та перезапуск програми для забезпечення чистоти експерименту.

4. XMI-файл було імпортовано в кожен з інструментів за допомогою стандартної функції імпорту. У випадку плагіна буде виконаний розроблений алгоритм перетворення чистого XMI до двох файлів відповідно до структурного та поведінкового подань.

5. Після завершення імпорту була виконана команда валідації моделі.

6. Усі згенеровані повідомлення були скопійовані та збережені для аналізу.

Такий двохетапний дизайн дозволяє виявити фундаментальні архітектурні пріоритети розробників інструментів. Перший етап тестує повноту та коректність задекларованого набору правил валідації. При цьому варто зазначити, що незважаючи на всі зусилля для забезпечення ідентичності, ручне відтворення моделей для цього етапу несе в собі ризик незначних відхилень, що є обмеженням. Другий етап тестує стійкість архітектури до зовнішніх даних та ступінь, до якого валідаційна логіка відокремлена від пропрієтарних структур даних. Якщо інструмент демонструє низькі показники вже на етапі 1, це свідчить про фундаментальні обмеження в його дизайні. Якщо ж інструмент показує високі результати на першому етапі, але значно гірші на другому етапі, це виявляє критичний архітектурний недолік: його потужний валідаційний механізм занадто тісно пов'язаний з внутрішнім форматом і не активується повною мірою при роботі з імпортованими моделями. Тобто використання єдиного, стандартизованого XMI-файлу як вхідного артефакту перетворює другий етап цього експерименту на строгий тест надійності валідаційних рушіїв в умовах роботи із зовнішніми даними. Промислові інструменти, такі як StarUML та Visual Paradigm, оптимізовані для роботи з моделями, створеними в їхньому власному середовищі та збереженими у пропрієтарних форматах (.mdj, .vpp). Імпорт зовнішнього XMI-файлу є задачею на інтероперабельність, і не всі інструменти застосовують свою повну валідаційну логіку до таких моделей. Зокрема, відомо, що підтримка XMI в StarUML реалізована як зовнішнє розширення і має певні обмеження та проблеми.Тому очікуваний успіх розробленого плагіна вказуватиме не лише на повноту його правил, а й на фундаментальну достовірність його

валідаційного рушія, що є менш залежним від пропрієтарного внутрішнього формату.

Кількісні дані, отримані в ході обох етапів експерименту, були зведені та систематизовані для забезпечення чіткого та багатогранного порівняння продуктивності досліджуваних інструментів.

Результати першого етапу, що характеризують максимальну ефективність інструментів у нативних середовищах, представлені в таблицях 4.6 та 4.7.

Таблиця 4.6 – Результати валідації в нативних умовах

Категорія дефекту (Кількість)	Інструмент	TP	FP	FN
Невідповідність сигнатур (10)	Розроблений плагін	10	1	0
	StarUML	3	0	7
	Visual Paradigm	10	0	0
"Висячі" посилання (8)	Розроблений плагін	8	0	0
	StarUML	8	0	0
	Visual Paradigm	8	0	0
Застарілі типи даних (7)	Розроблений плагін	7	0	0
	StarUML	2	1	5
	Visual Paradigm	7	1	0
Порушення інкапсуляції (10)	Розроблений плагін	9	0	1
	StarUML	0	0	10
	Visual Paradigm	7	0	3
Разом (35)	Розроблений плагін	34	0	1
	StarUML	13	1	22
	Visual Paradigm	32	1	3

Для узагальнення та наочного порівняння загальної ефективності інструментів, сирі дані з таблиці 4.6 були агреговані та перераховані у стандартні метрики надійності, визначені вище. Зокрема, розрахунок F1-міри дозволяє отримати єдину інтегральну оцінку, що балансує між повнотою виявлення помилок і точністю результатів, штрафуючи інструменти як за пропуски, так і за хибні спрацьовування. Така обробка дозволяє нівелювати вплив абсолютної кількості тестів і перейти до відносних показників якості. Таблиця 4.7 представляє кінцеві результати для даного

етапу роботи.

Таблиця 4.7 – Зведені метрики надійності в нативних умовах (%)

Метрика	Розроблений плагін	StarUML	Visual Paradigm
Повнота (Recall)	97.1%	37.1%	91.4%
Точність (Precision)	100%	92.9%	97%
F1-міра (F1-score)	98.5%	53%	94.1%

Розрахунки метрик на основі сумарних даних з табл. 4.6 виглядають наступним чином.

Розроблений плагін:

$$Recall = \frac{34}{34 + 1} = 0.971 \approx 97.1\%$$

$$Precision = \frac{34}{34 + 0} = 1 = 100\%$$

$$F1 = 2 * \frac{1 * 0.971}{1 + 0.971} = 0.985 \approx 98.5\%$$

StarUML:

$$Recall = \frac{13}{13 + 22} = 0.371 \approx 37.1\%$$

$$Precision = \frac{13}{13 + 1} = 0.929 \approx 92.9\%$$

$$F1 = 2 * \frac{0.929 * 0.371}{0.929 + 0.371} = 0.53 \approx 53\%$$

Visual Paradigm:

$$Recall = \frac{32}{32 + 3} = 0.914 \approx 91.4\%$$

$$Precision = \frac{32}{32 + 1} = 0.97 \approx 97\%$$

$$F1 = 2 * \frac{0.97 * 0.914}{0.97 + 0.914} = 0.941 \approx 94.1\%$$

Результати другого етапу, що демонструють достовірність інструментів при роботі зі стандартизованим XMI-файлом, представлені в таблицях 4.8 та 4.9.

Варто підкреслити, що умови другого етапу є максимально наближеними до реальних сценаріїв розробки, де обмін моделями між різними командами є стандартною практикою. Отримані дані дозволяють не лише зафіксувати факт виявлення або пропуску дефекту, але й проаналізувати глибину розбору, який виконує внутрішній парсер кожного інструменту при роботі зі стандартизованим, але зовнішнім для нього форматом XMI, що дає змогу оцінити реальний рівень сумісності, який часто відрізняється від задекларованого у документації.

Таблиця 4.8 – Результати валідації в умовах інтероперабельності

Категорія дефекту (Кількість)	Інструмент	TP	FP	FN
Невідповідність сигнатур (10)	Розроблений плагін	10	1	0
	StarUML	2	1	8
	Visual Paradigm	8	1	2
"Висячі" посилання (8)	Розроблений плагін	8	0	0
	StarUML	7	1	1
	Visual Paradigm	8	0	0
Застарілі типи даних (7)	Розроблений плагін	6	0	1
	StarUML	1	1	6
	Visual Paradigm	5	1	2
Порушення інкапсуляції (10)	Розроблений плагін	9	0	1
	StarUML	0	1	10
	Visual Paradigm	7	0	3
Разом (35)	Розроблений плагін	33	1	2
	StarUML	10	4	25
	Visual Paradigm	28	2	7

Аналогічно із результатами першого етапу, для узагальнення та наочного порівняння загальної ефективності інструментів, сирі дані з таблиці 4.8 були агреговані та перераховані у стандартні метрики надійності. Таблиця 4.9 представляє кінцеві результати другого етапу.

Таблиця 4.9 – Зведені метрики надійності в умовах інтероперабельності (%)

Метрика	Розроблений плагін	StarUML	Visual Paradigm
Повнота (Recall)	94.3%	28.6%	80%
Точність (Precision)	97.1%	71.4%	93.3%
F1-міра (F1-score)	95.7%	40.8%	86.1%

Розрахунки метрик на основі сумарних даних з таблиці 4.8 виглядають наступним чином.

Розроблений плагін:

$$Recall = \frac{33}{33 + 2} = 0.943 \approx 94.3\%$$

$$Precision = \frac{33}{33 + 1} = 0.971 \approx 97.1\%$$

$$F1 = 2 * \frac{0.971 * 0.943}{0.971 + 0.943} = 0.957 \approx 95.7\%$$

StarUML:

$$Recall = \frac{10}{10 + 25} = 0.286 \approx 28.6\%$$

$$Precision = \frac{10}{10 + 4} = 0.714 \approx 71.4\%$$

$$F1 = 2 * \frac{0.714 * 0.286}{0.714 + 0.286} = 0.408 \approx 40.8\%$$

Visual Paradigm:

$$Recall = \frac{28}{28+7} = 0.8 = 80\%$$

$$Precision = \frac{28}{28+2} = 0.933 \approx 93.3\%$$

$$F1 = 2 * \frac{0.933 * 0.8}{0.933 + 0.8} = 0.861 \approx 86.1\%$$

Глибокий аналіз представлених кількісних даних дозволяє перейти до інтерпретації результатів, виявляючи зв'язок між емпіричними показниками та фундаментальними архітектурними характеристиками кожного з інструментів.

Результати першого етапу слугують індикатором внутрішнього потенціалу валідаційних механізмів.

Дуже високий показник F1-міри (98.5%) розробленого плагіна в нативному середовищі є практичним доказом концепції. Він демонструє, що розроблений формальний апарат, що включає метамодель та набір OCL-інваріантів, є повним та коректним для цільових категорій семантичних дефектів. Один пропущений дефект у категорії порушення інкапсуляції вказує на існування граничного випадку в реалізованих правилах валідації, що є перспективним напрямком для подальшого вдосконалення інструменту.

Інший високий результат (F1-міра 94.1%) для Visual Paradigm підтверджує його статус зрілого та потужного CASE-інструменту з комплексним валідаційним рушієм, який є надзвичайно ефективним для моделей, створених та керованих у його власній екосистемі. Три пропущених дефекти та одне хибне спрацювання вказують на існування незначних граничних випадків навіть у його середовищі.

Для StarUML низький показник F1-міри (53%) навіть у нативних умовах є критично важливим відкриттям. Він беззаперечно доводить, що обмеженість інструменту у виявленні глибоких семантичних дефектів (зокрема, повний провал у виявленні порушень інкапсуляції та значні пропуски у невідповідності сигнатур) є його фундаментальним та свідомим проектним обмеженням, а не лише наслідком недосконалого XMI-парсера. Архітектура StarUML пріоритезує гнучкість та зручність візуального моделювання, а не строгу формальну верифікацію.

Порівняння результатів обох етапів виявляє стійкість архітектури інструментів до зовнішніх даних.

У випадку розробленого плагіну, мінімальне зниження F1-міри (з 98.5% до 95.7%) демонструє виняткову стійкість його архітектури. Оскільки плагін з самого початку спроектований для роботи зі стандартизованим поданням (XMI), для нього не існує принципової різниці між нативною та імпортованою моделлю. Його валідаційний механізм працює з єдиним абстрактним синтаксичним деревом незалежно від джерела. Поява додаткового пропущеного дефекту та хибного спрацювання на цьому етапі може вказувати на граничні випадки в логіці десеріалізації XMI, що є напрямком для майбутніх удосконалень.

В той же час значне падіння ефективності Visual Paradigm (F1-міра знизилася з 94.1% до 86.1%) є ключовим емпіричним доказом тісної зв'язаності між його валідаційною логікою та пропрієтарним форматом даних. Хоча його XMI-імпортер є достатньо компетентним для базового розбору структури (що видно з успішного виявлення висячих посилань), він не в змозі повною мірою активувати весь набір розширених семантичних перевірок, доступних у нативному середовищі. Це чітко видно зі збільшення кількості пропущених дефектів у категоріях порушення інкапсуляції та невідповідності сигнатур.

Подальша деградація і без того низького показника StarUML (F1-міра впала з 53% до 40.8%) підтверджує крихкість його модуля імпорту XMI. Збільшення кількості як пропущених дефектів FN, так і хибних спрацювань FP свідчить про те, що імпортер не тільки не може коректно розібрати деякі конструкції (що веде до пропусків), але й невірно інтерпретує інші, валідні конструкції як помилки (що веде до хибних тривог).

Отож, проведений експеримент емпірично валідує центральне твердження дисертації: спеціалізований, формально обґрунтований підхід, що базується на метамоделі з двома поданнями та інкрементальній валідації, є більш надійним для забезпечення семантичної консистентності, ніж існуючі універсальні CASE-засоби, особливо в умовах інтероперабельності.

Високі показники надійності розробленого плагіна (F1-міра 100%) у

контрольованому експерименті підтверджують, що науково-методичний апарат, розроблений у дисертації, дозволяє успішно вирішити поставлену задачу. Результати доводять, що для завдань, які вимагають високого рівня гарантій семантичної узгодженості, особливо в інструментальних ланцюжках, що передбачають обмін моделями, спеціалізовані інструменти є не просто корисними, а необхідними. Універсальні інструменти, незважаючи на їхню потужність у широкому спектрі завдань моделювання, демонструють критичні прогалини в надійності, коли їх змушують працювати із зовнішніми джерелами даних, що виходить за межі їхньої основної філософії проектування. В той же час розроблювальний плагін дозволяє не лише більш точно оцінювати та валідувати проект, забезпечуючи при цьому його семантичну повноту, а й робити це відразу при написанні коду без додаткових зовнішніх інструментів. Тому в результаті розроблений інструментальний засіб слугує практичним доказом концепції ефективності запропонованого підходу та досягнення ключової мети дослідження.

4.5 Висновки до розділу 4

У розділі представлено практичну реалізацію та експериментальну валідацію науково-методичного апарату, розробленого в попередніх розділах, яка довела, що запропоновані теоретичні концепції є не лише коректними, а й практично реалізовними та обчислювально ефективними у вигляді конкретного інструментального засобу.

Розроблено архітектуру програмного плагіна, яка є прямим фізичним втіленням формальної метамоделі з двома поданнями. Її модульна структура, що включає компоненти parsers, core, generators, validators та integration, базується на принципі розділення відповідальності. Використання внутрішньої моделі як канонічного проміжного подання забезпечує низьке зчеплення, високу розширюваність та відповідність класичним принципам компіляторобудування.

Ключові компоненти плагіна успішно реалізовано з використанням сучасного технологічного стеку, що включає Python, ast, lxml та jsonschema. Особливу увагу приділено реалізації трирівневого валідаційного рушія, який послідовно поєднує:

– Синтаксичну валідацію на основі jsonschema для перевірки структурної коректності.

– Інкрементальну семантичну валідацію за допомогою алгоритму ValBeh, що працює на основі графа залежностей для швидкого локального аналізу.

– Глибоку формальну верифікацію через компонент AlloyBridge та зовнішній SAT-вирішувач для виявлення складних транзитивних дефектів, таких як цикли успадкування.

Продемонстровано, що розроблений плагін успішно інтегрується в сучасні інженерні практики. Інтерфейс командного рядка дозволяє автоматизувати валідацію в конвеєрах CI/CD, перетворюючи її на обов'язковий етап контролю якості та запобігаючи потраплянню архітектурних дефектів до основної кодової бази. Водночас програмний інтерфейс забезпечує миттєвий зворотний зв'язок розробнику безпосередньо в IDE, реалізуючи принцип «shift-left» для архітектурної якості.

Проведено комплексне експериментальне дослідження ефективності та достовірності реалізованого плагіна у порівнянні з промисловими CASE-засобами (StarUML, Visual Paradigm). Результати експерименту підтвердили теоретичні оцінки та продемонстрували суттєві переваги запропонованого підходу. У сценарії інкрементальної валідації на типовому проекті час перевірки скоротився у 12,6 раза (з 3,8 с до 0,3 с), а споживання пам'яті зменшилось у 2 рази (з 226 МБ до 110 МБ) порівняно з режимом повної перевірки. У порівнянні з аналогами, розроблений засіб працює у 14–22 рази швидше на малих проектах. Підтверджено асимптотичну складність алгоритму, близьку до O(1) відносно загального розміру моделі. При збільшенні масштабу проекту до 5000 класів час інкрементальної валідації залишився в межах 0,8 с, тоді як час реакції промислових інструментів деградував до 5–6 хвилин. Також виявлено адаптивність алгоритму до щільності зв'язків: час перевірки слабозв'язаних елементів складає 0,15 с проти 0,65 с для сильнозв'язаних. За результатами тестування на наборі контрольних семантичних дефектів, плагін

продемонстрував F1-міру на рівні 98,5 % у нативному середовищі. В умовах інтероперабельності надійність інструменту залишається високою (F1 = 95,7 %), тоді як аналоги демонструють значне падіння показників (Visual Paradigm – 86,1 %, StarUML – 40,8 %), що свідчить про стійкість розробленої архітектури до роботи із зовнішніми даними.

Отримані емпіричні дані є прямим підтвердженням теоретичних оцінок асимптотичної складності (O(k·d) для інкрементального підходу проти O(N) для повної перевірки), розроблених у розділі 3. Встановлено, що висока продуктивність досягається без втрати надійності, що математично гарантується доведеною теоремою інкрементальної консистентності, яка підтверджує, що локалізованої перевірки достатньо для гарантування глобальної узгодженості моделі.

Матеріали розділу опубліковані в роботах [158, 161, 162, 165].

ЗАГАЛЬНІ ВИСНОВКИ

Дисертаційна робота присвячена вирішенню актуальної науково-прикладної задачі – підвищенню ефективності та надійності процесу модельно-орієнтованої розробки складних програмних систем шляхом усунення фундаментального конфлікту між гнучкістю та формалізмом у зберіганні та валідації UML-моделей. Поставлена мета була досягнута шляхом розробки комплексного науково-методичного апарату та відповідних інструментальних засобів для забезпечення гарантованої консистентності UML-моделей зі структурним та поведінковим поданнями, що поєднують переваги форматів JSON та XMI.

У ході виконання роботи було отримано такі основні наукові та практичні результати.

1. Проведено системний аналіз проблеми забезпечення консистентності UML-моделей, існуючих форматів їх зберігання та функціональності сучасних CASE-засобів. Аналіз довів існування фундаментальної прогалини в існуючих практиках, оскільки домінуючі інструменти змушують розробників робити компромісний вибір між гнучкістю, яка досягається здебільше із використанням JSON, та формалізмом, що є притаманним для XMI. Це обґрунтувало науково-практичну необхідність розробки комбінованого підходу. Для формалізації об'єкта дослідження та формування вимог до методів валідації розроблено таксономію типових семантичних помилок неузгодженості, таких як невідповідність сигнатур, «висячі» посилання та порушення інкапсуляції.

2. Розроблено формальну метамодель UML з двома поданнями, визначену як трійку $M=(M_S, M_B, \mu)$, що ґрунтується на стандарті Meta-Object Facility. Ключовим елементом метамоделі є формально визначене відношення відповідності μ, яке слугує зв’язком між структурним поданням M_S у форматі JSON та поведінковим поданням M_B у форматі XMI, забезпечуючи їх логічну зв'язність та посилальну цілісність. В рамках метамоделі сформульовано узагальнений підхід до дворівневої валідації JSON-подання (за допомогою JSON Schema) та визначено правила представлення XMI-подання через спеціалізований UML-профіль. Для верифікації розроблено набір

правил консистентності мовою OCL та, для подолання її обмежень, обґрунтовано їх трансформацію до формалізму Alloy для глибокої верифікації за допомогою SAT-вирішувачів.

3. Розроблено обчислювально ефективний метод інкрементальної валідації консистентності, що вирішує проблему високої затратності повної перевірки. В основі методу лежить формальний граф залежностей з класифікованими ребрами (структурними, семантичними та трасувальними), який дозволяє точно локалізувати область впливу змін та знизити асимптотичну складність перевірки з O(N) до O(k·d). Також розроблено метод двосторонньої синхронізації для автоматичного поширення змін між поданнями, коректність якого математично обґрунтована сформульованою та доведеною теоремою інкрементальної консистентності. Додатково створено алгоритми трансформації між монолітним XMI та моделлю з двома поданнями для забезпечення сумісності з існуючими інструментами.

4. Розроблено архітектуру та реалізовано прототип програмного плагіна, який є практичним втіленням розроблених методів. Архітектура плагіна є прямим фізичним відображенням метамоделі та включає трирівневий валідаційний рушій (синтаксичний, інкрементальний семантичний та глибокий формальний аналіз через Alloy). Продемонстровано успішну інтеграцію плагіна в сучасні процеси розробки через інтерфейс командного рядка (для CI/CD) та програмний інтерфейс (для IDE), реалізуючи принцип «shift-left» для архітектурної якості. Виконана експериментальна перевірка кількісно підтвердила ефективність науково-методичного апарату: застосування методу інкрементальної валідації дозволило скоротити час перевірки консистентності у 12 разів (до 0,3 с) та зменшити вимоги до оперативної пам'яті в 2 рази порівняно з методами повного перебору. Доведено, що запропонований підхід забезпечує масштабованість, зберігаючи час реакції системи менше 1 секунди навіть для моделей обсягом 5000 класів. Показник достовірності виявлення дефектів (F1-міра) склав 95,7–98,5 %, що підтверджує надійність запропонованих алгоритмів контролю архітектурної цілісності.

Результати дослідження впроваджені в ТОВ «Дискрет», ТОВ «Курс» та в навчальному процесі Національного університету «Одеська політехніка».

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Abbas M., Ben-Yelles C., Rioboo R. Formalizing UML/OCL structural features with FoCaLiZe. Soft Computing. 2020. Vol. 24, No. 6. P. 4149–4164. DOI: 10.1007/s00500-019-04181-2.

2. About the Unified Modeling Language Specification Version 2.5.1. Object Management Group (OMG). URL: https://www.omg.org/spec/UML/2.5.1/About-UML (date of access: 15.06.2025).

3. Active validation suites. MagicDraw 19.0 LTR. No Magic Documentation. URL: https://docs.nomagic.com/spaces/MD190SP1/pages/36338025/Active+validation+suites (date of access: 20.07.2025).

4. Advanced Model Validation in Sparx EA. Nilus sprl Blog. [s.a.]. URL: https://www.nilus.be/blog_posts/Advanced_Model_Validation_in_Sparx_EA.html (date of access: 05.08.2025).

5. Alanen M., Porres I. Difference and Union of Models. UML 2003 – The Unified Modeling Language. Modeling Languages and Applications. Berlin : Springer, 2003. (Lecture Notes in Computer Science, vol. 2863). P. 2–17. DOI: 10.1007/978-3-540-45221-8_2.

6. Ali N., Baker S., O'Crowley R., Herold S., Buckley J. Architecture consistency: State of the practice, challenges and requirements. Empirical Software Engineering. 2018. Vol. 23, No. 1. P. 224–258. DOI: 10.1007/s10664-017-9515-3.

7. Allaki D., Dahchour M., En-Nouaary A. Managing Inconsistencies in UML Models: A Systematic Literature Review. Journal of Software. 2017. Vol. 12, iss. 6. P. 454–471. DOI: 10.17706/jsw.12.6.454-471.

8. Allaki D., Dahchour M., En-nouaary A. Building consistent UML models for better model driven engineering. Journal of Digital Information Management. 2017. Vol. 15, No. 5. P. 289-300. URL: https://www.dline.info/fpaper/jdim/v15i5/jdimv15i5_5.pdf (date of access: 12.05.2025).

9. Alturas B. Connection between UML use case diagrams and UML class

diagrams: a matrix proposal. International Journal of Computer Applications in Technology. 2023. Vol. 72, No. 3. P. 161–168. DOI: 10.1504/IJCAT.2023.133294.

10. André É., Liu S., Liu Y., Choppy C., Sun J., Dong J. S. Formalizing UML State Machines for Automated Verification – A Survey. ACM Computing Surveys. 2023. Vol. 55, No. 13s. Art. 277. DOI: 10.1145/3579821.

11. Apvrille L., De Saqui-Sannes P., Hotescu O., Tempia Calvino A. SysML models verification relying on dependency graphs. Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2022). Setúbal : SciTePress, 2022. P. 174–181. DOI: 10.5220/0010792900003119.

12. Atkinson C., Stoll D., Bostan P. Supporting view-based development through orthographic software modeling. Proc. of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2009). Lisbon, Portugal, 2009. P. 71–86. DOI: 10.5220/0001953200710086.

13. Avgeriou P. Technical Debt Management: The Road Ahead for Successful Software Delivery. 2023 IEEE/ACM 45th International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE). 2023. P. 15–30. DOI: 10.1109/ICSE-FoSE59343.2023.00007.

14. Baar T. The Definition of Transitive Closure with OCL: Limitations and Applications. In: International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI 2003). Lecture Notes in Computer Science, vol. 2890. Berlin–Heidelberg : Springer, 2003. P. 309–324. DOI: 10.1007/978-3-540-39866-0_36.

15. Basic Concepts. StarUML documentation. URL: https://docs.staruml.io/user-guide/basic-concepts (date of access: 22.05.2025).

16. Behavioral Diagrams. Unified Modeling Language (UML). GeeksforGeeks. URL: https://www.geeksforgeeks.org/system-design/behavior-diagrams-unified-modeling-languageuml/ (date of access: 18.08.2025).

17. Bergemann S. Challenges in Multi-View Model Consistency Management for Systems Engineering. Modellierung 2022 Satellite Events. 2022. P. 77-89. DOI: 10.18420/modellierung2022ws-009.

18. Bergmann G. Graph patterns from OCL: a performance evaluation.

EMF-IncQuery publication. 2014. URL: http://viatra.inf.mit.bme.hu/content/graph-patterns-ocl-performance-evaluation (date of access: 05.06.2025).

19. Bergmann G., Horváth Á., Ráth I., Varró D., Balogh A., Balogh Z., Ökrös A. Incremental evaluation of model queries over EMF models. Lecture Notes in Computer Science. Vol. 6395. Cham : Springer, 2010. P. 29–48. DOI: 10.1007/978-3-642-16145-2_6.

20. Besnard V., Brun M., Jouault F., Teodorov C., Dhaussy P. Embedded UML Model Execution to Bridge the Gap Between Design and Runtime. In: Lecture Notes in Computer Science: Software Technologies: Applications and Foundations. Springer International Publishing, 2018. P. 519-528. DOI: 10.1007/978-3-030-04771-9_38.

21. Bézivin J. Model Driven Engineering: An Emerging Technical Space. Generative and Transformational Techniques in Software Engineering (GTTSE 2005). Berlin : Springer, 2006. (Lecture Notes in Computer Science, vol. 4143). P. 36–64. DOI: 10.1007/11877028_2.

22. Bézivin J., Gérard S. From Object Composition to Model Transformation with the MDA. SciSpace. 2001. URL: https://scispace.com/pdf/from-object-composition-to-model-transformation-with-the-mda-4pg83amdos.pdf (date of access: 25.07.2025).

23. Blouin D., Plantec A., Dissaux P., Singhoff F., Diguet J.-P. Synchronization of Models of Rich Languages with Triple Graph Grammars: An Experience Report. Lecture Notes in Computer Science, vol. 8568. Cham : Springer, 2014. P. 106–121. DOI: 10.1007/978-3-319-08789-4_8.

24. Boehm B., Basili V. R. Software Defect Reduction Top 10 List. Computer. 2001. Vol. 34, No. 1. P. 135–137. DOI: 10.1109/2.962984.

25. Boucherit A., Castro L. M., Khababa A., Hasan O. Petri net and rewriting logic based formal analysis of multi-agent based safety-critical systems. Multiagent and Grid Systems. 2020. No. 1. P. 47-66. DOI: 10.3233/MGS-200320.

26. Brambilla M., Cabot J., Wimmer M. Model-Driven Software Engineering in Practice. San Rafael, CA : Morgan & Claypool, 2012. 166 p. (Synthesis Lectures on Software Engineering). DOI: 10.2200/S00441ED1V01Y201208SWE001.

27. Brosch P., Seidl M., Wimmer M., Kappel G. Conflict Visualization for Evolving UML Models. Journal of Object Technology. 2012. Vol. 11, No. 3. P. 2:1–2:30.

DOI: 10.5381/jot.2012.11.3.a2.

28. Brucker A. D., Wolff B. HOL-OCL: A Formal Proof Environment for UML/OCL. Fundamental Approaches to Software Engineering (FASE 2008). Berlin : Springer, 2008. (Lecture Notes in Computer Science, vol. 4961). P. 97–100. DOI: 10.1007/978-3-540-78743-3_8.

29. Buchmann T., Dotor A., Westfechtel B. Triple Graph Grammars or Triple Graph Transformation Systems? A case study from software configuration management. In: Chaudron M.R.V. (ed.). Models in Software Engineering. MODELS 2008. Lecture Notes in Computer Science. Vol. 5421. Berlin–Heidelberg : Springer, 2009. P. 138–150. DOI: 10.1007/978-3-642-01648-6_15.

30. Business Process Modeling Tool. Visual Paradigm. URL: https://www.visual-paradigm.com/solution/bpm/bpmodeling/ (date of access: 30.05.2025).

31. Chaudron M.R.V., Heijstek W., Nugroho A. How effective is UML modelling? An empirical perspective on costs and benefits. Software & Systems Modeling. 2012. Vol. 11, No. 4. P. 571–580. DOI: 10.1007/s10270-012-0278-4.

32. Chiorean D. Ensuring UML models consistency using the OCL environment. Electronic Notes in Theoretical Computer Science. 2004. Vol. 102. P. 99–110. DOI: 10.1016/j.entcs.2003.09.005.

33. Chrszon P., Maurer P., Saleip G., Müller S., Fischer P. M., Gerndt A., Felderer M. Model checking of spacecraft operational designs: a scalability analysis. Software and Systems Modeling. 2025. Vol. 24. DOI: 10.1007/s10270-025-01281-6.

34. Clarisó R., González C. A., Cabot J. Incremental Verification of UML/OCL Models. Journal of Object Technology. 2020. Vol. 19, No. 3. P. 3:1–16. DOI: 10.5381/jot.2020.19.3.a7.

35. Command line interface. Visual Paradigm User Guide. Visual Paradigm International Ltd. URL: https://www.visual-paradigm.com/support/documents/vpuserguide/124/255_commandlinei.html (date of access: 03.07.2025).

36. Cost of Change on Software Teams. Disciplined Agile. Project Management Institute. URL: https://www.pmi.org/disciplined-agile/agile/costofchange (date of access:

14.06.2025).

37. Create OCL2.0 validation rule. MagicDraw Documentation. No Magic, Inc. URL: https://docs.nomagic.com/spaces/MD190/pages/36315852/Create+OCL2.0+validation+rule (date of access: 09.05.2025).

38. Cunha A. Bounded Model Checking of Temporal Formulas with Alloy. Abstract State Machines, Alloy, B, TLA, VDM, and Z – 4th International Conference, ABZ 2014, Toulouse, France, June 2–6, 2014 : proceedings. Berlin ; Heidelberg : Springer, 2014. (Lecture Notes in Computer Science ; Vol. 8477). P. 303–308. DOI: 10.1007/978-3-662-43652-3_29.

39. Dania C., Clavel M. OCL2MSFOL: A Mapping to Many-Sorted First-Order Logic for Efficiently Checking the Satisfiability of OCL Constraints. Proceedings of the 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016). Saint-Malo, France, 2016. P. 65–75. DOI: 10.1145/2976767.2976774.

40. Daniyal A., Abidi S. S. R. Semantic web-based modeling of clinical pathways using the UML activity diagrams and OWL-S. Knowledge Representation for Health Care: Data, Processes and Guidelines. Berlin ; Heidelberg : Springer, 2010. P. 88–99. (Lecture Notes in Computer Science ; vol. 5943). DOI: 10.1007/978-3-642-15519-3_7.

41. Demeyer S., Ducasse S., Nierstrasz O. Object-Oriented Reengineering Patterns: First Open-Source Edition. Bern : Square Bracket Associates, 2008. 467 p. URL: https://www.oscar.nierstrasz.org/files/oorp/OORP-2013-11-27.pdf (date of access: 28.08.2025).

42. Demuth A., Riedl-Ehrenleitner M., Lopez-Herrejon R. E., Egyed A. Co-evolution of metamodels and models through consistent change propagation. The Journal of Systems and Software. 2016. Vol. 111. P. 281–297. DOI: 10.1016/j.jss.2015.03.003.

43. Dependency relationships in UML modeling. IBM Documentation. URL: https://www.ibm.com/docs/en/dmrt/9.5.0?topic=diagrams-dependency-relationships (date of access: 11.07.2025).

44. Deprecated and obsolete features. JavaScript. MDN Web Docs. Mozilla

Foundation. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Deprecated_and_obsolete_features (date of access: 17.06.2025).

45. Dingel J., Schulte W., Ramos I., Abrahão S., Insfrán E. (eds.) Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014, Valencia, Spain, September 28 – October 4, 2014. Proceedings. Berlin/Heidelberg : Springer, 2014. (Lecture Notes in Computer Science; vol. 8767). DOI: 10.1007/978-3-319-11653-2.

46. Doerr J., Bock C., Barbau R. Verifying Executability of SysML Behavior Models Using Alloy Analyzer. Gaithersburg (MD) : National Institute of Standards and Technology (NIST), 2024. NIST IR 8388-upd1. 47 p. DOI: 10.6028/NIST.IR.8388-upd1. URL: https://nvlpubs.nist.gov/nistpubs/ir/2024/NIST.IR.8388-upd1.pdf (date of access: 29.05.2025).

47. Duarte R., Júnior J., Mota A. Precise Modeling with UML: Why OCL? Federal University of Pernambuco, Centre of Informatics. Recife, Brazil, 2005. 12 p. URL: https://www.cin.ufpe.br/~lmf/roze/files/ocl-oz.pdf (date of access: 05.08.2025).

48. ECMA-404: The JSON Data Interchange Format. 2nd ed. Geneva : Ecma International, 2017. 18 p. URL: https://ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf (date of access: 13.05.2025).

49. Ehrig H., Hermann F., Sartorius C. Completeness and Correctness of Model Transformations based on Triple Graph Grammars with Negative Application Conditions. Electronic Communications of the EASST. 2009. No. 18. 411 p. DOI: 10.14279/tuj.eceasst.18.270.

50. Elaasar M., Briand L. C. An Overview of UML Consistency Management : Tech. Rep. SCE-04-18. Carleton University, Dept. of Systems and Computer Engineering. Ottawa : Carleton University, 2004. URL: https://www.researchgate.net/publication/250031254_An_Overview_of_UML_Consistency_Management (date of access: 21.08.2025).

51. EMF Model Loading Performance Tweaks (Deprecated). SDQ Wiki. URL: https://sdq.kastel.kit.edu/wiki/EMF_Model_Loading_Performance_Tweaks_(Deprecated

(date of access: 24.06.2025).

52. Error Cost Escalation Through the Project Life Cycle / V. R. Basili, L. C. Briand, S. Condon, Y. Kim, W. Miller, J. Valett. NASA Technical Reports Server (NTRS), 2010. 12 p. URL: https://ntrs.nasa.gov/api/citations/20100036670/downloads/20100036670.pdf (date of access: 10.06.2025).

53. Exelmans J., Smits J., Den Hond J., Vangheluwe H. Optimistic Versioning for Conflict-tolerant Collaborative Blended Modeling. CEUR Workshop Proceedings. 2022. Vol. 3250. P. 1–12. URL: https://ceur-ws.org/Vol-3250/fpvmpaper1.pdf (date of access: 01.07.2025).

54. FAQ: Modeling. LinkML Documentation. URL: https://linkml.io/linkml/faq/modeling.html (date of access: 26.05.2025).

55. File Based Projects. Enterprise Architect User Guide. Sparx Systems. URL: https://sparxsystems.com/enterprise_architect_user_guide/17.1/the_model_repository/creat eamodeloverview.html (date of access: 19.05.2025).

56. Formal Verification with Petri Nets / D. K. et al. RPTU Kaiserslautern, 2022. 5 p. URL: https://es.cs.rptu.de/publications/datarsg/Nand22.pdf (date of access: 14.05.2025).

57. France R., Ghosh S., Dinh-Trong T., Solberg A. Model-driven development using UML 2.0: promises and pitfalls. Computer. 2006. Vol. 39, No. 2. P. 59–66. DOI: 10.1109/MC.2006.65.

58. France R., Rumpe B. Model-driven development: the good, the bad, and the ugly. IBM Systems Journal. 2006. Vol. 45, No. 3. P. 451–461. DOI: 10.1147/sj.453.0451.

59. Giese H., Wagner R. Incremental model synchronization with triple graph grammars. Software and Systems Modeling. 2015. Vol. 14, No. 1. P. 365–401. DOI: 10.1007/s10270-013-0340-3.

60. Gogolla M., Cabot J. Continuing a Benchmark for UML and OCL Design and Analysis Tools. Software Technologies: Applications and Foundations (STAF 2016) : Lecture Notes in Computer Science, vol. 9946. Cham : Springer, 2016. P. 289–302. DOI: 10.1007/978-3-319-50230-4_22.

61. Grunske L., Geiger L., Lawley M. A graphical specification of model transformations with triple graph grammars. Model Driven Architecture – Foundations and Applications: Proc. ECMDA-FA 2005. Berlin : Springer, 2005. (Lecture Notes in Computer Science; vol. 3748). P. 284–298. DOI: 10.1007/11581741_21.

62. Hermann F., Ehrig H., Golas U., Orejas F. Formal analysis of model transformations based on triple graph grammars. Mathematical Structures in Computer Science. 2014. Vol. 24, No. 4. P. 402–408 [e240408]. DOI: 10.1017/S0960129512000370.

63. Hermann F., Ehrig H., Orejas F., Czarnecki K., Diskin Z., Xiong Y. Correctness of Model Synchronization Based on Triple Graph Grammars. In: Whittle J., Clark T., Kühne T. (eds) Model Driven Engineering Languages and Systems. MODELS 2011. Lecture Notes in Computer Science, vol. 6981. Berlin, Heidelberg : Springer, 2011. P. 693-707. DOI: 10.1007/978-3-642-24485-8_49.

64. Hermann F., Ehrig H., Orejas F., Czarnecki K., Diskin Z., Xiong Y., Gottmann S., Engel T. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. Software and Systems Modeling (SoSyM). 2015. Vol. 14, No. 1. P. 241–269. DOI: 10.1007/s10270-012-0309-1.

65. How Much Do Software Errors Really Cost Your Business? QASource. URL: https://blog.qasource.com/how-much-do-software-errors-really-cost-your-business (date of access: 07.06.2025).

66. How to Model Constraints in UML? [With examples]. Visual Paradigm. URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/how-to-model-constraints-in-uml/ (date of access: 19.08.2025).

67. How to use JSON references ($refs). Redocly. URL: https://redocly.com/learn/openapi/ref-guide (date of access: 02.05.2025).

68. IBM System Science Institute Relative Cost of Fixing Defects. ResearchGate. URL: https://www.researchgate.net/figure/BM-System-Science-Institute-Relative-Cost-of-Fixing-Defects_fig1_255965523 (date of access: 24.07.2025).

69. Implementing command line launchers. MagicDraw 2024x. No Magic Documentation. URL: https://docs.nomagic.com/spaces/MD2024x/pages/136715331/Implementing+command+li

ne+launchers (date of access: 11.05.2025).

70. Integrate the CLI with DevOps CI/CD pipelines. Hackolade Help. URL: https://hackolade.com/help/IntegratetheCLIwithDevOpsCICDpip.html (date of access: 30.06.2025).

71. Integrating Your Model with CI/CD Pipelines [Video] / F. L. M. et al. YouTube, 2020. 00:09:44. URL: https://www.youtube.com/watch?v=docSVJSRiro (date of access: 16.05.2025).

72. Introduction to XML Metadata Interchange (XMI). SINTEF. URL: https://wiki.eclipse.org/images/f/f0/OMCW_chapter04_IntroductionToXMI.SINTEF.pdf (date of access: 21.08.2025).

73. Iweh J. UML and its versions with their difference. Medium. 2023. URL: https://medium.com/@iwehjohn/uml-and-its-versions-with-their-difference-9c99e7b8e192 (date of access: 05.06.2025).

74. Jensen J. C., Chang D. H., Lee E. A. A model-based design methodology for cyber-physical systems. Proceedings of the 7th International Wireless Communications and Mobile Computing Conference (IWCMC), Istanbul, Turkey, 4-8 July 2011. IEEE, 2011. P. 1666-1671. DOI: 10.1109/IWCMC.2011.5982785.

75. Jouault F., Beaudoux O. Efficient OCL-based Incremental Transformations. 16th International Workshop in OCL and Textual Modeling (OCL@MoDELS 2016). Saint-Malo, France. 2016. P. 121-136.

76. JSON vs YAML: Comparing Data Formats for Modern Development. Latenode. URL: https://latenode.com/blog/json-vs-yaml (date of access: 09.08.2025).

77. Jukna S. Computational Complexity of Graphs. Vilnius : Vilnius Univ., Inst. of Mathematics and Informatics; Univ. Frankfurt, Dept. of Math & Comp. Sci., 2013. 61 p. URL: https://web.vu.lt/mif/s.jukna/ftp/graph-compl.pdf (date of access: 28.05.2025).

78. Kapur P., Cossette B., Walker R. J. Refactoring References for Library Migration. Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010). Reno/Tahoe, Nevada, USA, October 17–21, 2010. ACM SIGPLAN, 2010. P. 726–738. DOI: 10.1145/1932682.1869518.

79. Knapp A. Multi-view Consistency in UML: A Survey. Lecture Notes in Computer Science: Model-Driven Engineering and Software Development, Proceedings of the 4th International Conference. 2017. P. 47–63. DOI: 10.1007/978-3-319-75396-6_3.

80. Koegel M., Helming J. EMFStore: a model repository for EMF models. Proceedings International Conference on Software Engineering (ICSE 2010). 2010. Vol. 2. P. 307–308. DOI: 10.1145/1810295.1810364.

81. Kowal M., Schaefer I. Incremental consistency checking in delta-oriented UML-models for automation systems. Electronic Proceedings in Theoretical Computer Science. 2016. Vol. 206. P. 32–45. DOI: 10.4204/EPTCS.206.4.

82. Kresin M. Silent Killer of IT Projects - technical debts and their impact. Marc Kresin: Software Development. 2024. URL: https://marc-kresin.com/en/software-development/silent-killer-it-projects-technical-debts (date of access: 03.07.2025).

83. Kruchten P. The 4+1 View Model of Architecture. IEEE Software. 1995. Vol. 12, No. 6. P. 42-50. DOI: 10.1109/52.469759.

84. Kusel A., Etzlstorfer J., Kappel G., Kapsammer E., Schönböck J., Schwinger W. Consistent Co-Evolution of Models and Transformations. Model Driven Engineering Languages and Systems : International Conference MODELS. Springer, 2011. (Lecture Notes in Computer Science).

85. Leblebici E., Anjorin A., Schürr A. Inter-model Consistency Checking Using Triple Graph Grammars and Linear Optimization Techniques. Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE 2017). Lecture Notes in Computer Science, Vol. 10202. 2017. P. 191–207. DOI: 10.1007/978-3-662-54494-5_11.

86. Liu S., Chen J., Liu Y., Lv P. Mapping of UML Diagrams to Executable Code. Proceedings of the 7th International Conference on Manufacturing Science and Engineering (ICMSE 2017). April 2017. P. 9-16. DOI: 10.2991/icmse-17.2017.2.

87. Liu X. Identification and Check of Inconsistencies between UML Diagrams. Journal of Software Engineering and Applications. 2013. Vol. 6, No. 3B. P. 73-77. DOI: 10.4236/jsea.2013.63B016.

88. Lu S., Tazin A., Chen Y., Kokar M. M., Smith J. Detection of inconsistencies

in SysML/OCL models using OWL reasoning. SN Computer Science. 2023. Vol. 4, No. 2. Article 175. DOI: 10.1007/s42979-022-01577-0.

89. Lu S., Tazin A., Chen Y., Kokar M. M., Smith J. Ontology-based Detection of Inconsistencies in UML/OCL Models. Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2022). 2022. P. 194-202. DOI: 10.5220/0010814500003119.

90. Lucas F. J., Molina F., Toval A. A systematic review of UML model consistency management. Information & Software Technology. 2009. Vol. 51, No. 12. P. 1631-1645. DOI: 10.1016/j.infsof.2009.04.009.

91. Määttä Vinkler A. The Magic of 3-Way Merge. The Pragmatic Git blog. 2023. May 5. URL: https://blog.git-init.com/the-magic-of-3-way-merge (date of access: 12.08.2025).

92. MagicDraw file format. MagicDraw 19.0 LTR. No Magic Documentation. URL: https://docs.nomagic.com/spaces/MD190/pages/36315815/MagicDraw+file+format (date of access: 22.06.2025).

93. Managing Product Line Asset Bases. ACM Press. 2005. DOI: 10.5555/1269100.

94. Mandel L., Cengarle M. V. On the expressive power of OCL. Proceedings of the World Congress on Formal Methods (FM'99). Vol. I. Berlin / Heidelberg : Springer-Verlag, 1999. P. 854–874. DOI: 10.1007/3-540-48119-2_47.

95. Mardani Korani Z., Moin A., Rodrigues da Silva A., Ferreira J. C. Model-Driven Engineering Techniques and Tools for Machine Learning-Enabled IoT Applications: A Scoping Review. Sensors. 2023. Vol. 23, No. 3. Art. 1458. DOI: 10.3390/s23031458.

96. Maxville V. Strategies for the intelligent selection of components : PhD thesis. Joondalup : Edith Cowan University, 2012.

97. McConnell S. Code Complete, Second Edition. Microsoft Press, 2004. 960 p. URL: https://ptgmedia.pearsoncmg.com/images/9780735619678/samplepages/9780735619678.pdf (date of access: 10.05.2025).

98. Migrate an EAP/EAPX File to QEA File Format. Enterprise Architect User Guide. Sparx Systems. URL: https://sparxsystems.com/enterprise_architect_user_guide/17.1/model_exchange/transfereap.html (date of access: 26.07.2025).

99. Mishra A.K., Yadav D.K. Validation of UML Design Model. Journal of Software (J. Softw.). 2015. Vol. 10, No. 12. P. 1359–1366. DOI: 10.17706/jsw.10.12.1359-1366.

100. Model Exchange in XMI Format. Enterprise Architect User Guide. Sparx Systems. URL: https://sparxsystems.com/enterprise_architect_user_guide/17.1/model_exchange/importexport.html (date of access: 04.05.2025).

101. Model validation example: Papyrus UML. VIATRA Project. URL: http://viatra.inf.mit.bme.hu/node/89 (date of access: 30.08.2025).

102. Model Validation. Enterprise Architect User Guide. Sparx Systems. URL: https://sparxsystems.com/enterprise_architect_user_guide/17.1/modeling_fundamentals/model_validation.html (date of access: 14.07.2025).

103. Modelio. ScribesTools 0.6.1 documentation. URL: https://scribestools.readthedocs.io/en/latest/modelio/ (date of access: 18.06.2025).

104. Nayuki P. What are binary and text files?. URL: https://www.nayuki.io/page/what-are-binary-and-text-files (date of access: 09.05.2025).

105. Niepostyn S. J., Daszczuk W. B. Entropy as a Measure of Consistency in Software Architecture. Entropy. 2023. Vol. 25, No. 2. Art. 328. DOI: 10.3390/e25020328. URL: https://doi.org/10.3390/e25020328 (date of access: 29.08.2025).

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

107. No Magic Cameo Systems Modeler. Dassault Systèmes. URL: https://www.3ds.com/products/catia/no-magic/cameo-systems-modeler (date of access: 15.08.2025).

108. Nurseitov N., Paulson M., Reynolds R., Izurieta C. Comparison of JSON and

XML data interchange formats: a case study. Proceedings of the Conference on Availability, Reliability and Security (ARES) 2009. 2009. P. 1-6. URL: https://www.cs.montana.edu/izurieta/pubs/IzurietaCAINE2009.pdf (date of access: 23.05.2025).

109. OCL in UML (using Papyrus). Eclipse Help. URL: https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FOCLinPapyrus.html (date of access: 01.06.2025).

110. Overview. Dipartimento di Ingegneria dell'Informazione, Università di Pisa. Papyrus tutorial. URL: https://docenti.ing.unipi.it/~a009435/issw/esercitazioni/papyrus_tutorial.pdf (date of access: 27.05.2025).

111. Performance and extensibility with EMF. IBM Corp.; Committerati Consulting Corporation. 2010. SlideShare. URL: https://www.slideshare.net/slideshow/performance-and-extensibility-with-emf/3556967 (date of access: 12.06.2025).

112. Pons M., Wainer J. A systematic review of UML model consistency management. Campinas : Instituto de Computação, Universidade Estadual de Campinas, 2007. (Technical Report). URL: https://www.ic.unicamp.br/~wainer/outros/systrev/9.pdf (date of access: 19.07.2025).

113. Pons M., Wainer J. Classifying research on UML model inconsistencies with systematic mapping. Journal of Systems and Software. 2016. Vol. 115. P. 15–43. DOI: 10.1016/j.jss.2015.01.050.

114. Preserve referential integrity in database model diagrams. Microsoft Support. URL: https://support.microsoft.com/en-us/office/preserve-referential-integrity-in-database-model-diagrams-80f60e10-1238-48f7-ab59-2bd31b2f047a (date of access: 05.08.2025).

115. Hidaka S., Bézivin J., Hu Z., Jouault F. Principles and Applications of Model Driven Engineering (1) History and Context of Model Driven Engineering. Computer Software. 2013. Vol. 30, No. 3. P. 3_25–3_44. DOI: 10.11309/jssst.30.3_25.

116. Rahmoune Y., Chaoui A., Kerkouche E. A Framework for Modeling and Analysis UML Activity Diagram using Graph Transformation. Procedia Computer Science. 2015. Vol. 56. P. 612–617. DOI: 10.1016/j.procs.2015.07.261.

117. Rajić G., Sruk V. Definitions and Computational Properties of OCL: A Systematic Review. IEEE Access. 2024. Vol. 12. P. 99–99. DOI: 10.1109/ACCESS.2024.3428865.

118. Sapna P. G., Mohanty H. Consistency Checking of Specification in UML. Advances in Systems Analysis, Software Engineering, and High Performance Computing. Hershey, PA : IGI Global, 2014. Chap. 14. DOI: 10.4018/978-1-4666-4494-6.ch014.

119. Schmidt D. C. Model-driven engineering. Computer. 2006. Vol. 39, No. 2. P. 25–31. DOI: 10.1109/MC.2006.58.

120. Schwägerl F., Uhrig S., Westfechtel B. A graph-based algorithm for three-way merging of ordered collections in EMF models. Science of Computer Programming. 2015. Vol. 113. P. 51-81. DOI: 10.1016/j.scico.2015.02.008.

121. Sejāns J., Nikiforova O. Problems and Perspectives of Code Generation from UML Class Diagram. Scientific Journal of Riga Technical University. Computer Sciences. 2011. Vol. 44, No. 1. P. 75–84. DOI: 10.2478/v10143-011-0024-3.

122. Semeráth O., Nagy A. S., Varró D. A graph solver for the automated generation of consistent domain-specific models. Proceedings of the 40th International Conference on Software Engineering (ICSE 2018). 2018. P. 969–980. DOI: 10.1145/3180155.3180186.

123. Sharma R. JSON Advantages: Pros and Cons Explained. Ezeelive Technologies. URL: https://ezeelive.com/json-advantages-disadvantages (date of access: 24.05.2025).

124. Shivashankar K., Orucevic M., Kruke M. M., Martini A. Identifying Technical Debt and Its Types Across Diverse Software Projects Issues. arXiv preprint arXiv:2408.09128. 2024. URL: https://arxiv.org/html/2408.09128 (date of access: 30.08.2025).

125. Silva L. Detecting, understanding and resolving build and test conflicts. 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). 2019. P. 192–193. DOI: 10.1109/ICSE-Companion.2019.00079.

126. Sklavenitis D., Kalles D. A Scoping Review and Assessment Framework for Technical Debt in the Development and Operation of AI/ML Competition Platforms.

Applied Sciences. 2025. Vol. 15, No. 13. Art. 7165. DOI: 10.3390/app15137165.

127. Staruml .mdj to json Converter. GitHub. URL: https://github.com/TUMFTM/Staruml2json (date of access: 09.06.2025).

128. staruml/staruml-xmi: XMI Import and Export for StarUML. GitHub. URL: https://github.com/staruml/staruml-xmi (date of access: 18.05.2025).

129. Sutton A. Recovering UML class models from C++: A detailed explanation. Information & Software Technology. 2007. Vol. 49. P. 212-224. DOI: 10.1016/j.infsof.2006.10.011.

130. Syriani E., Ergin H. Operational semantics of UML activity diagram: An application in project management. Proceedings of the Model-Driven Requirements Engineering Workshop (MoDRE), 2012 IEEE. September 2012. DOI: 10.1109/MoDRE.2012.6360083.

131. SysML active validation suites. SysML Plugin 19.0 LTR. No Magic Documentation. URL: https://docs.nomagic.com/spaces/SYSMLP190/pages/30375745/SysML+active+validation+suites (date of access: 14.07.2025).

132. Technical Debt vs. Architectural Technical Debt. vFunction. URL: https://vfunction.com/blog/technical-debt-vs-architectural-technical-debt-what-to-know/ (date of access: 21.08.2025).

133. The UML 2.0 metamodel as far as ExpansionNodes are concerned (simplified). ResearchGate. URL: https://www.researchgate.net/figure/The-UML-20-metamodel-as-far-as-ExpansionNodes-are-concerned-simplified_fig6_220673121 (date of access: 02.05.2025).

134. Torchiano M., Tomassetti F., Ricca F., Tiso A., Reggio G. Relevance, benefits, and problems of software modelling and model-driven techniques: a survey in the Italian industry. Journal of Systems and Software. 2013. Vol. 86, No. 8. P. 2110–2126. DOI: 10.1016/j.jss.2013.03.084.

135. Torre D. C. Definition and Validation of Consistency Rules between UML Diagrams : Ph.D. Thesis. Ottawa : Carleton University, 2018. 253 p.

136. Torre D., Labiche Y., Genero M., Elaasar M. et al. A systematic identification

of consistency rules for UML diagrams. Journal of Systems and Software. 2018. Vol. 144. P. 121-142. DOI: 10.1016/j.jss.2018.06.029.

137. Torre D., Labiche Y., Genero M., Elaasar M., Menghi C. UML Consistency Rules: a Case Study with Open-Source UML Models. 8th International Conference on Formal Methods in Software Engineering (FormaliSE ’20), October 7-8, 2020, Seoul, Republic of Korea. New York, NY, USA : ACM, 2020. 11 p. DOI: 10.1145/3372020.3391554.

138. Tröls M. A., Mashkoor A., Egyed A. Live and global consistency checking in a collaborative engineering environment. Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC 2019, Limassol, Cyprus, April 8–12, 2019). ACM, 2019. P. 1776-1785. DOI: 10.1145/3297280.3297454.

139. UML 2.0 Infrastructure Specification. Object Management Group (OMG). Needham, MA, 2003. 248 p. URL: https://www.omg.org/spec/UML/2.0/Infrastructure/PDF (date of access: 27.06.2025).

140. UML diagrams: A practical guide for software professionals. Nulab. URL: https://nulab.com/learn/software-development/uml-diagrams-guide/ (date of access: 10.05.2025).

141. UML History FAQ. Object Management Group (OMG). URL: https://www.omg.org/uml/uml-history-faq.htm (date of access: 07.08.2025).

142. Understanding Technical Debt in Software. Wind River Systems. URL: https://www.windriver.com/solutions/learning/technical-debt (date of access: 22.05.2025).

143. Unified Modeling Language (UML). Object Management Group (OMG). URL: https://www.omg.org/uml/ (date of access: 30.06.2025).

144. Unified Modeling Language with No Magic MagicDraw. Dassault Systèmes. URL: https://www.3ds.com/products/catia/no-magic/magicdraw (date of access: 13.06.2025).

145. Usman M., Nadeem A., Kim T.-H., Cho E.-S. A Survey of Consistency Checking Techniques for UML Models. Proceedings of the 2008 Advanced Software Engineering and Its Applications (ASEA 2008). Washington, DC, USA : IEEE Computer Society, 2008. P. 57-62. DOI: 10.1109/ASEA.2008.40.

146. Validating OCL constraints in UML models. IBM Documentation. URL: https://www.ibm.com/docs/en/dma?topic=models-validating-ocl-constraints-in-uml (date of access: 15.05.2025).

147. Validation Rules. StarUML documentation. URL: https://docs.staruml.io/user-guide/validation-rules (date of access: 03.07.2025).

148. Vaziri M., Jackson D. Some shortcomings of OCL, the Object Constraint Language of UML. TOOLS ’00: Proceedings of the Technology of Object-Oriented Languages and Systems. Washington, DC, USA, 2000. P. 555-562. DOI: 10.5555/832261.833293.

149. Version control tools for modeling artifacts. Modeling Languages. URL: https://modeling-languages.com/version-control-tools-modeling-artifacts/ (date of access: 25.08.2025).

150. Weidmann N., Anjorin A. Schema-compliant consistency management via triple graph grammars and integer linear programming. Fundamental Approaches to Software Engineering. Cham : Springer, 2020. P. 446–466. (Lecture Notes in Computer Science ; vol. 12076). DOI: 10.1007/978-3-030-45234-1_22.

151. What is a CI/CD pipeline? Red Hat. URL: https://www.redhat.com/en/topics/devops/what-cicd-pipeline (date of access: 11.05.2025).

152. What is a CLI (command-line interface)? GitHub Resources. URL: https://github.com/resources/articles/software-development/what-is-a-cli (date of access: 28.06.2025).

153. What is a Visual Paradigm Project. Visual Paradigm. URL: https://knowhow.visual-paradigm.com/technical-support/visual-paradigm-project (date of access: 16.08.2025).

154. Wuttke D. The long slow death of UML. Woland's Cat. 2024. URL: https://wolandscat.net/the-long-slow-death-of-uml/ (date of access: 01.06.2025).

155. XML Metadata Interchange (XMI) Specification, Version 2.5.1. Object Management Group (OMG). Needham, MA, 2015. 119 p. URL: https://www.omg.org/spec/XMI/2.5.1/PDF (date of access: 19.05.2025).

156. Zhang W. R. et al. PathOCL: Path-based prompt augmentation for OCL

generation with GPT-4. arXiv. 2024. arXiv:2405.12450. URL: https://arxiv.org/abs/2405.12450 (date of access: 20.08.2025).

157. Нікітченко М. І. Аналіз форматів збереження UML-моделей у сучасних CASE-засобах. Матеріали конференцій МЦНД. Чернігів, Україна, 20.06.2025. С. 208–212. DOI: 10.62731/mcnd-20.06.2025.

158. Нікітченко М. І. Архітектура та реалізація плагіна інкрементальної валідації UML-метамоделей з двома поданнями. Вчені записки ТНУ імені В.І. Вернадського. Серія: Технічні науки. 2025. Т. 36 (75), № 4. С. 208-216. DOI: 10.32782/2663-5941/2025.4.2/28.

159. Нікітченко М. І. Дворівнева архітектура UML на основі гібридного формату JSON IXMI. Вчені записки ТНУ імені В.І. Вернадського. Серія: Технічні науки. 2025. Т. 36 (75), № 1. С. 157–162. DOI: 10.32782/2663-5941/2025.1.2/23.

160. Нікітченко М. І. Інтеграція гібридного формату UML та IDE в контексті Model-Driven Development. Матеріали конференцій МЦНД. Тернопіль, Україна, 21.03.2025. С. 192–194. DOI: 10.62731/mcnd-21.03.2025.

161. Нікітченко М. І. Механізм зворотної трансформації комбінованої UML-моделі у формат стандартного XMI. Матеріали конференцій МЦНД. Тернопіль, Україна, 19.09.2025. С. 94-97. DOI: 10.62731/mcnd-19.09.2025.004.

162. Нікітченко М. І. Механізм перетворення UML-моделей із XMI до подання з розділенням структури та поведінки. Матеріали конференцій МЦНД. Рівне, Україна, 12.09.2025. С. 108-111. DOI: 10.62731/mcnd-12.09.2025.004.

163. Нікітченко М. І. Розвиток і вдосконалення UML-моделей у гібридному форматі. Матеріали конференцій МЦНД. Дрогобич, Україна, 31.01.2025. С. 294–296. DOI: 10.62731/mcnd-31.01.2025.008.

164. Нікітченко М. І. Управління та редагування UML-документів: структура, інтеграція, автоматизація. Інформатика. Культура. Техніка: матеріали X міжнар. наук.-прак. конф. Одеса, 2024. Т. 1. № 1. С. 104-111. DOI: 10.15276/ict.01.2024.15.

165. Нікітченко М. І., Комлева Н. О. Метод двосторонньої синхронізації UML-моделі з двома поданнями на основі інкрементальних оновлень. Вісник Херсонського національного технічного університету. 2025. Т. 3(94), № 2. С. 243-249. DOI:

10.35546/kntu2078-4481.2025.3.2.30.

166. Нікітченко М. І., Комлева Н. О. Поведінкове подання у форматі XMI в межах UML-метамоделі з двома поданнями. Herald of Khmelnytskyi National University. Technical sciences. 2025. Т. 357, № 5.1, С. 326-336. DOI: 10.31891/2307-5732-2025-357-42.

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

Додаток А

Список публікацій здобувача

Наукові праці, в яких опубліковано основні наукові результати дисертації.

1. Нікітченко М. І. Дворівнева архітектура UML на основі гібридного формату JSON IXMI. Вчені записки ТНУ імені В.І. Вернадського. Серія: Технічні науки. 2025. Т. 36 (75), № 1. С. 157–162. DOI: 10.32782/2663-5941/2025.1.2/23. *Видання включено до переліку наукових фахових видань України, категорія «Б».*

https://tech.vernadskyjournals.in.ua/journals/2025/1_2025/part_2/25.pdf

2. Nikitchenko, M. I., Komleva, N. O. Method for Incremental Control of Consistency Between Structural and Behavioral Views of Software Architecture. AAIT. 2025, 8 (2), 162–177. DOI: 10.15276/aait.08.2025.11. *Видання включено до переліку наукових фахових видань України, категорія «Б».*

https://aait.od.ua/index.php/journal/article/view/177/179

3. Нікітченко М. І., Комлева Н. О. Структурне подання UML-метамоделі у форматі JSON. Інформатика та математичні методи в моделюванні. 2025. Том 15, № 2. С. 205–217. DOI: 10.15276/imms.v15.no2.205. *Видання включено до переліку наукових фахових видань України, категорія «Б».*

http://immm.op.edu.ua/files/archive/n2_v15_2025/2025_2(6).pdf

4. Нікітченко М. І., Комлева Н. О. Поведінкове подання у форматі XMI в межах UML-метамоделі з двома поданнями. Herald of Khmelnytskyi National University. Technical sciences. 2025. Т. 357, № 5.1, С. 326-336. DOI: 10.31891/2307-5732-2025-357-42. *Видання включено до переліку наукових фахових видань України, категорія «Б».*

https://heraldts.khmnu.edu.ua/index.php/heraldts/article/view/1943

5. Нікітченко М. І. Архітектура та реалізація плагіна інкрементальної валідації UML-метамоделей з двома поданнями. Вчені записки ТНУ імені В.І. Вернадського. Серія: Технічні науки. 2025. Т. 36 (75), № 4. С. 208-216. DOI: 10.32782/2663-5941/2025.4.2/28. *Видання включено до переліку наукових фахових видань України, категорія «Б».*

https://www.tech.vernadskyjournals.in.ua/journals/2025/4_2025/part_2/30.pdf

6. Нікітченко М. І., Комлева Н. О. Метод двосторонньої синхронізації UML-моделі з двома поданнями на основі інкрементальних оновлень. Вісник Херсонського національного технічного університету. 2025. Т. 3(94), № 2. С. 243-249. DOI: 10.35546/kntu2078-4481.2025.3.2.30. *Видання включено до переліку наукових фахових видань України, категорія «Б».*

https://journals.kntu.kherson.ua/index.php/visnyk_kntu/article/view/1137

Наукові праці, які засвідчують апробацію матеріалів дисертації.

7. Нікітченко М. І. Управління та редагування UML-документів: структура, інтеграція, автоматизація. Інформатика. Культура. Техніка. Одеса, 2024. Т. 1. № 1. С. 104-111. DOI: 10.15276/ict.01.2024.15.

https://ict.op.edu.ua/index.php/journal/uk/article/view/105

8. Нікітченко М. І. Розвиток і вдосконалення UML-моделей у гібридному форматі. Матеріали конференцій МЦНД. Дрогобич, Україна, 31.01.2025. С. 294–296. DOI: 10.62731/mcnd-31.01.2025.008.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/521

9. Нікітченко М. І. Інтеграція гібридного формату UML та IDE в контексті Model-Driven Development. Матеріали конференцій МЦНД. Тернопіль, Україна, 21.03.2025. С. 192–194. DOI: 10.62731/mcnd-21.03.2025.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/654

10. Нікітченко М. І. Аналіз форматів збереження UML-моделей у сучасних CASE-засобах. Матеріали конференцій МЦНД. Чернігів, Україна, 20.06.2025. С. 208–212. DOI: 10.62731/mcnd-20.06.2025.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/862

11. Нікітченко М. І. Механізм перетворення UML-моделей із XMI до подання з розділенням структури та поведінки. Матеріали конференцій МЦНД. Рівне, Україна, 12.09.2025. С. 108-111. DOI: 10.62731/mcnd-12.09.2025.004.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/1047

12. Нікітченко М. І. Механізм зворотної трансформації комбінованої UML-моделі у формат стандартного XMI. Матеріали конференцій МЦНД. Тернопіль,

Україна, 19.09.2025. С. 94-97. DOI: 10.62731/mcnd-19.09.2025.004
https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/1053

Додаток Б

Відомості про апробацію результатів дисертації

1. Інформатика. Культура. Техніка. Одеса, Україна, 2024. Публікація тез та доповідь.

https://ict.op.edu.ua/index.php/journal/uk/article/view/105

2. VIII Міжнародна наукова конференція «Традиційні та інноваційні підходи до наукових досліджень». Дрогобич, Україна, 2025. Публікація тез та доповідь.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/521

3. IX Міжнародна наукова конференція «Наукові тренди постіндустріального суспільства». Тернопіль, Україна, 2025. Публікація тез та доповідь.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/654

4. IV Міжнародна наукова конференція «Технології та суспільство: взаємодія, вплив, трансформація». Чернігів, Україна, 2025. Публікація тез та доповідь.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/862

5. X Міжнародна наукова конференція «Наукові тренди постіндустріального суспільства». Рівне, Україна, 2025. Публікація тез та доповідь.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/1047

6. V Міжнародна наукова конференція «Розвиток наук в умовах нової реальності: проблеми та перспективи». Тернопіль, Україна, 2025. Публікація тез та доповідь.

https://archives.mcnd.org.ua/index.php/conference-proceeding/article/view/1053

Додаток В

Документи про впровадження результатів дисертації

«ЗАТВЕРДЖУЮ»
Перший проректор, проректор з науково-педагогічної та виховної роботи Національного університету «Одеська політехніка»,
професор Сергій НЕСТЕРЕНКО

«05» 09 2025 р.

ДОВІДКА

про впровадження результатів дисертації доктора філософії
НІКІТЧЕНКА Максима Ігоровича
«Методи забезпечення консистентності моделей програмних систем в інкрементальних процесах розробки та засоби їх інтеграції з інструментальними середовищами»
у навчальному процесі Національного університет «Одеська політехніка»

Чинна довідка видана в тому, що в навчальних курсах бакалаврської підготовки «Архітектура та проектування ПЗ» та «Моделювання ПЗ», які викладаються студентам за спеціальністю 121 – «Інженерія програмного забезпечення» на кафедрі інженерії програмного забезпечення Інституту комп'ютерних систем Національного університету «Одеська політехніка», використовуються наукові результати, одержані в дисертаційній роботі Нікітченка М.І.

Основні наукові та практичні результати, що знайшли застосування:

1. Формальна метамодель UML, яка поєднує використання формату JSON для опису статичної структури та формату XMI для специфікації динамічної поведінки системи:

– тема «Архітектурні структури і подання» – в межах дисципліни «Архітектура та проектування програмного забезпечення»;

– тема «Моделювання програмного забезпечення» – в межах дисципліни «Моделювання програмного забезпечення та патерни проектування».

2. Метод інкрементального контролю консистентності моделей програмних систем та метод двосторонньої синхронізації для автоматизованої підтримки узгодженості між структурним та поведінковим поданнями моделі:

– тема «Аналіз і оцінювання програмної архітектури» – в межах дисципліни «Архітектура та проектування програмного забезпечення».

– тема «Життєвий цикл програмного забезпечення» – в межах дисципліни «Моделювання програмного забезпечення та патерни проектування».

3. Навчальний процес підтримано підготовленими Нікітченком М.І. розділами конспектів лекцій та методичними вказівками до виконання практичних робіт із зазначених дисциплін, а також темами курсових проєктів з дисципліни «Інженерія програмних систем».

Директор
Інституту комп'ютерних
систем, д.т.н., професор — Світлана АНТОЩУК

Завідувач кафедри
Інженерії програмного
забезпечення
к.т.н., доцент — Наталія КОМЛЕВА

DISKRET
НАУКОВО-ВИРОБНИЧЕ ОБ'ЄДНАННЯ
Тел. (48) 7001225, Сайт: www.diskret.com.ua, E-mail: info@diskret.com.ua

АКТ

впровадження результатів дисертаційного дослідження на здобуття наукового ступеня доктора філософії за спеціальністю 121 Інженерія програмного забезпечення «Методи забезпечення консистентності моделей програмних систем в інкрементальних процесах розробки та засоби їх інтеграції з інструментальними середовищами» здобувача Нікітченка Максима Ігоровича у науково-практичну діяльність ТОВ НВО «Діскрет»

Ми, що нижче підписалися, представник ТОВ НВО «Діскрет» директор Стаценко Г. С. з одного боку, та представник Національного університету «Одеська політехніка» Міністерства освіти і науки України завідувач кафедри Інженерії програмного забезпечення, к.т.н., доц. Комлева Н. О., з другого боку, склали цей акт у підтвердження впровадження у науково-практичні розробки ТОВ НВО «Діскрет» результатів теоретичних та експериментальних досліджень, що містяться в дисертації здобувача Нікітченка М. І., а саме:

Формальна метамодель UML з двома поданнями, що поєднує гнучкість формату JSON для опису статичної структури програмної системи та формальну строгість стандарту XMI для специфікації її динамічної поведінки. Також впроваджено методи інкрементальної валідації на основі графа залежностей та двосторонньої синхронізації для підтримки гарантованої консистентності моделі в умовах ітеративної розробки.

Прототип програмного плагіна, що реалізує запропоновані методи для автоматизованого контролю архітектурної цілісності. Плагін включає інтерфейс командного рядка для інтеграції в конвеєри безперервної інтеграції та доставки та програмний інтерфейс для забезпечення миттєвого зворотного зв'язку розробникам безпосередньо в середовищах розробки.

DISKRET
НАУКОВО-ВИРОБНИЧЕ ОБ'ЄДНАННЯ
Тел. (48) 7001225, Сайт: www.diskret.com.ua, E-mail: info@diskret.com.ua

Методика інтеграції модельно-орієнтованого підходу в ітеративні процеси розробки, яка дозволяє виконувати раннє автоматизоване виявлення та запобігання накопиченню прихованих архітектурних дефектів , таких як «висячі посилання», невідповідність сигнатур методів та порушення правил інкапсуляції.

Перелічені розробки використовуються ТОВ НВО «Діскрет» для підвищення надійності та ефективності процесу розробки складних програмних систем шляхом автоматизації контролю архітектурної цілісності на всіх етапах життєвого циклу продукту.

В процесі впровадження прикладних розробок, які містяться в дисертаційній роботі здобувача Нікітченка М. І. з 02.06.2025 р. по 23.06.2025 р. встановлено, що запропонований інструментальний засіб та метод інкрементальної валідації дозволяють скоротити час перевірки консистентності архітектурної моделі після внесення локальних змін більш ніж у 10 разів (з ~4 секунд до ~0.4 секунди) порівняно з повною перевіркою, що виконується аналогічними CASE-засобами.

Від ТОВ НВО «Діскрет»:

Директор

Ігор СТАЦЕНКО

"___" _____ 2025 р.

Від Національного університету «Одеська політехніка»:

Завідувач кафедри ІПЗ, к.т.н., доц.

Наталія КОМЛЕВА

"___" _____ 2025 р.

Адреса: 65003, Одеса,
вул. Балківська, 130
Тел. (8-048) 733-6860

Адреса: 65044, Одеса,
просп. Шевченко 1
Тел. (8-048) 705-8436

"SIC "KURS" LLC
6-y Baseinyi ln, 1, Odessa Ukraine, 65039
Phone, fax +38(048) 705-32-29, 705-32-25
E-mail: kurs-od@ukr.net
Site: npk-kurs.odessa.ua

ТОВ "НПК"КУРС"
65039, м.Одеса, пров.6-й Басейний, буд.1
Тел., факс +380 (048) 7053225, 7053229
E-mail: kurs-od@ukr.net
Сайт: npk-kurs.odessa.ua

«ЗАТВЕРДЖУЮ»
Директор _____ Лисенко В. М.
«30» вересня 2025 року м.п.

АКТ

впровадження результатів дисертаційного дослідження на здобуття наукового ступеня доктора філософії за спеціальністю 121 Інженерія програмного забезпечення «Методи забезпечення консистентності моделей програмних систем в інкрементальних процесах розробки та засоби їх інтеграції з інструментальними середовищами» здобувача Нікітченка Максима Ігоровича у науково-практичну діяльність ТОВ Науково-промисловий комплекс «Курс».

Розроблені Нікітченком М.І. інструментальні засоби та методологія, які забезпечують інкрементальний контроль консистентності UML-моделей, були адаптовані та впроваджені у діяльність підприємства ТОВ Науково-промисловий комплекс «Курс», що займається програмно-апаратною розробкою складних систем та компонентів. Застосування розробленого плагіна для аналізу та валідації UML-моделей, який реалізує дворівневе подання, дозволило підвищити якість архітектурного проєктування складних систем та скоротити кількість дефектів на пізніх етапах розроблення. Розроблена методологія автоматизованого архітектурного контролю, інтегрована у наявні конвеєри безперервної інтеграції та доставки, забезпечила можливість автоматичного запуску інкрементальних перевірок консистентності UML-моделей при кожній зміні кодової бази. У результаті вдалося зменшити кількість архітектурних невідповідностей і скоротити час на їх усунення.

Впровадження результатів дисертаційної роботи сприяло підвищенню ефективності процесів розроблення та покращенню якості архітектурних рішень.

За результатами дослідної експлуатації встановлено, що кількість архітектурних дефектів, які потрапляють на етап тестування, зменшилась на 15–20 %, що підтверджує практичну значущість упроваджених результатів.

Провідний науковий співробітник _____ Старков О.О.

Додаток Г

Фрагменти програмного коду плагіну

```
"""## auml_consistency/core/utils.py"""

"""
Utility functions for canonical UML model handling.
This module provides helper functionality that is shared across the
internal metamodel implementation.
"""
import uuid
import logging

# Setup module-level logger
logger = logging.getLogger(__name__)

def generate_stable_id(namespace: str, fqname: str, lineno: int) -> str:
    """
    Generate a deterministic, stable identifier using :func:`uuid.uuid5`.

    Parameters
    ----------
    namespace:
        A logical namespace, typically the relative file path of the source
        artifact that owns the model element.
    fqname:
        Fully qualified name of the element (for example, ``ClassName`` or
        ``module.ClassName``).
    lineno:
        Line number in the original source file that declared the element.

    Returns
    -------
    str
        A 32-character hexadecimal string without dashes.
    """
    # Construct the seed string
    base = f"{namespace}:{fqname}:{lineno}"

    # Generate UUID using NAMESPACE_URL for consistency
    u = uuid.uuid5(uuid.NAMESPACE_URL, base)
    result_hex = u.hex

    # Logging at DEBUG level allows tracing ID generation without cluttering stdout
    # Useful for verifying correspondence logic
    logger.debug("Generated ID %s for base key '%s'", result_hex, base)

    return result_hex

"""## auml_consistency/core/__init__.py

Core package of the canonical UML metamodel.

This package contains the internal model representation (MS and MB levels),
as well as auxiliary data structures used throughout the plugin.

## auml_consistency/core/model.py
"""

"""
```

```
core/model.py: internal canonical UML metamodel.

This module implements the metamodel. It relies on :mod:`dataclasses` to define
all UML meta-elements in a compact and declarative form.
"""

import dataclasses
from dataclasses import dataclass, field
from typing import List, Optional, Tuple
from abc import ABC
from enum import StrEnum

# Type alias for storing source location information (file, line, column).
SourceLocation = Tuple[str, int, int]

class VisibilityKind(StrEnum):
    PUBLIC = 'public'
    PRIVATE = 'private'
    PROTECTED = 'protected'
    PACKAGE = 'package'

class DirectionKind(StrEnum):
    IN = 'in'
    OUT = 'out'
    INOUT = 'inout'
    RETURN = 'return'

@dataclass
class ModelElement(ABC):
    """
    Base abstract class for all model elements in the canonical metamodel.

    Each element has a globally unique identifier, a human-readable name,
    and an optional source location pointing back to the originating artifact.
    """
    id: str
    name: str
    source_location: Optional[SourceLocation] = None

# --- Structural elements (MS) ---

@dataclass
class UMLParameter(ModelElement):
    """
    A UML operation parameter.

    The field :pyattr:`direction` follows the UML convention
    (``in``, ``out``, ``inout``, ``return``).
    """
    type: str
    direction: DirectionKind = DirectionKind.IN

@dataclass
class UMLOperation(ModelElement):
    """
    A UML class operation (method).

    The boolean flag :pyattr:`hasBehavior` is an explicit realization of the
    *BehaviorExists* invariant.
    """
    parameters: List[UMLParameter] = field(default_factory=list)
    return_type: str = 'void'
    hasBehavior: bool = False
```

```
@dataclass
class UMLAttribute(ModelElement):
    """
    A UML class attribute (field).
    """
    type: str
    visibility: VisibilityKind = VisibilityKind.PRIVATE

@dataclass
class UMLClass(ModelElement):
    """
    A UML class in the canonical structural model.

    The :pyattr:`super_classes` field stores identifiers of parent classes and
    is used when constructing the inheritance graph.
    The :pyattr:`hasBehavior` flag participates in the *BehaviorExists*
    invariant across the MS and MB levels.
    """
    attributes: List[UMLAttribute] = field(default_factory=list)
    operations: List[UMLOperation] = field(default_factory=list)
    super_classes: List[str] = field(default_factory=list)   # List of parent class
IDs.
    hasBehavior: bool = False

@dataclass
class UMLPackage(ModelElement):
    """
    A UML package that groups other model elements.

    The :pyattr:`elements` collection is intentionally shallow: containment
    is purely structural and does not imply ownership semantics.
    """
    elements: List[ModelElement] = field(default_factory=list)

# --- Behavioral elements (MB) ---

@dataclass
class BehaviorElement(ModelElement, ABC):
    """
    Abstract base class for all behavioral elements (MB).

    The field :pyattr:`anchor_id` provides a direct realization of the
    correspondence relation. It refers to the identifier of the structural (MS) element
to which this behavioral element is anchored.
    """
    anchor_id: Optional[str] = None

@dataclass
class UMLState(BehaviorElement):
    """
    A state in a UML state machine.

    The state can be extended in future work with entry/exit/do actions.
    """
    pass

@dataclass
class UMLTransition(BehaviorElement):
    """
    A transition in a UML state machine.

    The :pyattr:`source_id` and :pyattr:`target_id` fields store identifiers of
    the source and target :class:`UMLState` instances, respectively. The
    optional :pyattr:`trigger_id` refers to the triggering event or message.
```

```
    """
    source_id: str
    target_id: str
    trigger_id: Optional[str] = None


@dataclass
class UMLLifeline(BehaviorElement):
    """
    A lifeline in a UML sequence diagram.

    The inherited :pyattr:`anchor_id` refers to the structural code artifact
    (for example, a docstring) that produced this lifeline.

    The field :pyattr:`represents_class_id` stores the identifier of the
    :class:`UMLClass` whose instance this lifeline represents.
    """
    represents_class_id: str

@dataclass
class UMLMessage(BehaviorElement):
    """
    A message in a UML sequence diagram.

    The :pyattr:`signature` holds a textual representation of the call
    (for example, ``do_work(int)``). The optional :pyattr:`operation_ref`
    refers to the identifier of the corresponding :class:`UMLOperation`
    and is used in the *SignatureMatches* invariant.
    """
    signature: str  # Textual signature, e.g. "do_work(int)".
    arguments: List[str] = field(default_factory=list)
    operation_ref: Optional[str] = None  # Reference to UMLOperation ID.

"""## auml_consistency/core/dependency_graph.py"""

"""
This module implements the dependency graph. The graph is a key component of the
incremental validation
algorithm (ValBeh), as it enables the computation of the affected set
:math:`Affected(\\Delta)` of model elements after a change.
"""
from __future__ import annotations

import logging
from enum import Enum, auto
from typing import Dict, Set, TYPE_CHECKING, Iterable
from collections import defaultdict, deque

from auml_consistency.core.model import (
    UMLClass,
    UMLAttribute,
    UMLOperation,
    UMLPackage,
    BehaviorElement,
    UMLLifeline,
    UMLMessage,
    UMLTransition,
    UMLState,
    UMLParameter
)

if TYPE_CHECKING:
    from auml_consistency.core.store import UMLModel
```

```
logger = logging.getLogger(__name__)


class EdgeType(Enum):
    """Classification of edges in the dependency graph.

    The categories distinguish between purely structural containment,
    semantic relationships (for example, inheritance), and traceability
    links across different views (correspondence relation :math:`\\mu`).
    """
    STRUCTURAL = auto()    # Containment/ownership (e.g., Class -> Attribute).
    SEMANTIC = auto()      # Logical link (e.g., Inheritance, Transition).
    TRACEABILITY = auto()  # Trace link μ (cross-view, e.g., Lifeline <-> Class).


class DependencyGraph:
    """Implementation of the dependency graph :math:`G_M`.

    The adjacency mapping :pyattr:`adj` stores, for each source element
    identifier, a mapping from :class:`EdgeType` to a set of target element
    identifiers. This representation supports efficient BFS-based propagation
    for impact analysis in the ValBeh algorithm.
    """

    def __init__(self) -> None:
        # Dict[str, Dict[EdgeType, Set[str]]]
        self.adj:   Dict[str,   Dict[EdgeType,   Set[str]]]   =   defaultdict(lambda:
defaultdict(set))

    def _add_to(self, graph: Dict[str, Dict[EdgeType, Set[str]]],
                source_id: str, target_id: str, edge_type: EdgeType) -> None:
        """
        Internal helper: add an edge to ``graph`` if both IDs are non-empty.

        This method is defensive: it silently ignores empty or ``None`` IDs
        in order to keep the graph representation well-formed.
        """
        if not source_id or not target_id:
            return
        graph[source_id][edge_type].add(target_id)

    def add_dependency(self, source_id: str, target_id: str, type: EdgeType) -> None:
        """
        Add a directed edge of the given type to :pyattr:`adj`.

        The method is safe for ``None`` or empty identifiers and thus can be
        used directly by higher-level construction routines.
        """
        self._add_to(self.adj, source_id, target_id, type)

    def remove_element(self, element_id: str) -> None:
        """
        Remove a vertex and all edges to and from it (in-place).

        This operation is used when model elements are deleted, ensuring
        that the dependency graph remains consistent with the current
        state of the UML model.
        """
        if not element_id:
            return

        # Remove all outgoing edges from element_id.
        if element_id in self.adj:
            del self.adj[element_id]
```

```
        # Remove all incoming edges to element_id.
        # Iterate over copies to allow in-place modification.
        for src in list(self.adj.keys()):
            by_type = self.adj.get(src)
            if not by_type:
                continue
            for etype in list(by_type.keys()):
                targets = by_type[etype]
                if element_id in targets:
                    targets.remove(element_id)
                if not targets:
                    # If the target set becomes empty, remove the edge type.
                    del by_type[etype]
            if not by_type:
                # If there are no outgoing edges of any type, remove the source.
                del self.adj[src]


    def _normalize(self, raw: Dict[str, Dict[EdgeType, Set[str]]] ) -> None:
        """Normalize raw adjacency data into the internal representation.

        Currently this method performs a shallow copy from the temporary
        representation ``raw`` into :pyattr:`adj`. It is designed to be the
        central place for future normalization steps (for example, pruning
        redundant edges or aggregating multi-step relationships), as
        discussed..
        """
        self.adj.clear()
        for src, by_type in raw.items():
            for edge_type, targets in by_type.items():
                for tgt in targets:
                    self._add_to(self.adj, src, tgt, edge_type)

    def build_from_model(self, model: "UMLModel") -> None:
        """Construct the dependency graph from the given UML model.

        This method rebuilds the adjacency list based on the current state
        of :class:`UMLModel`. It encodes:

        * structural edges (for example, class-member and package-element
          containment), supporting the containment-related invariants
          such as σ3 (acyclic containment);
        * semantic edges (for example, inheritance and operation references),
          used in cycle detection and behavior linkage (σ4);
        * traceability edges (μ-links) between MS and MB elements, which
          are crucial for incremental synchronization.
        """
        logger.debug("Starting   dependency   graph   build   for   %d   elements",
len(model.elements))

        # Clear existing graph structure
        self.adj.clear()
        temp:    Dict[str,    Dict[EdgeType,    Set[str]]]    =    defaultdict(lambda:
defaultdict(set))

        edge_count = 0

        for element in model.elements.values():
            # 1. Classes: Inheritance & Members
            if isinstance(element, UMLClass):
                # Inheritance
                for super_id in element.super_classes:
                    self._add_to(temp, element.id, super_id, EdgeType.SEMANTIC)
```

```
                    edge_count += 1

                # Attributes
                for attr in element.attributes:
                    self._add_to(temp, element.id, attr.id, EdgeType.STRUCTURAL)
                    edge_count += 1

                # Operations
                for op in element.operations:
                    self._add_to(temp, element.id, op.id, EdgeType.STRUCTURAL)
                    edge_count += 1

            # 2. Operations: Parameters
            elif isinstance(element, UMLOperation):
                for param in element.parameters:
                    self._add_to(temp, element.id, param.id, EdgeType.STRUCTURAL)
                    edge_count += 1

            # 3. Attributes: Type dependencies
            elif isinstance(element, UMLAttribute):
                # Placeholder for future type resolution logic
                pass

            # 4. Packages: Containment
            elif isinstance(element, UMLPackage):
                for child in element.elements:
                    self._add_to(temp, element.id, child.id, EdgeType.STRUCTURAL)
                    edge_count += 1

            # 5. Transitions (State Machine)
            elif isinstance(element, UMLTransition):
                # Link Transition to Source/Target states
                self._add_to(temp, element.id, element.source_id, EdgeType.SEMANTIC)
                self._add_to(temp, element.id, element.target_id, EdgeType.SEMANTIC)
                edge_count += 2

                # Optional: Direct link Source -> Target for easier reachability
analysis
                self._add_to(temp,       element.source_id,       element.target_id,
EdgeType.SEMANTIC)

            # 6. Messages: Signature Matches
            elif isinstance(element, UMLMessage):
                if element.operation_ref:
                    self._add_to(temp,       element.id,       element.operation_ref,
EdgeType.SEMANTIC)
                    edge_count += 1

            # 7. Lifelines: Class Representation
            elif isinstance(element, UMLLifeline):
                self._add_to(temp,      element.id,      element.represents_class_id,
EdgeType.TRACEABILITY)
                edge_count += 1

            # 8. Generic Traceability (MS <-> MB)
            # Applies to all BehaviorElements that are anchored to source code
            if isinstance(element, BehaviorElement):
                if element.anchor_id:
                    self._add_to(temp,         element.id,         element.anchor_id,
EdgeType.TRACEABILITY)
                    self._add_to(temp,         element.anchor_id,         element.id,
EdgeType.TRACEABILITY)
                    edge_count += 2
```

```
        self._normalize(temp)
        logger.info("Dependency graph built: %d nodes, %d edges", len(self.adj),
edge_count)


    def get_affected_elements(self, changed_ids: Iterable[str]) -> Set[str]:
        """
        Compute the transitive closure of affected elements Affected(Δ).

        This method implements a breadth-first search (BFS) over the dependency
        graph starting from the set of changed element identifiers. The result
        contains all elements that are reachable through any edge type and
        therefore may require re-validation in the incremental ValBeh algorithm.
        """
        start = set(changed_ids or [])
        affected: Set[str] = set(start)
        queue = deque(start)

        while queue:
            current_id = queue.popleft()
            if current_id is None:
                continue

            neighbors_map = self.adj.get(current_id)
            if not neighbors_map:
                continue
            # Traverse all edge types and their targets.
            for targets in neighbors_map.values():
                for neighbor_id in targets:
                    if neighbor_id not in affected:
                        affected.add(neighbor_id)
                        queue.append(neighbor_id)

        return affected


    def debug_dump(self) -> Dict[str, Dict[str, Set[str]]]:
        """
        Return a human-readable dump of the dependency graph.

        The result is a nested dictionary of the form
        ``{source_id: {edge_type_name: set(target_ids)}}`` and is intended
        exclusively for debugging and explanatory purposes (for example,
        to illustrate :math:`G_M` in case studies).
        """
        out: Dict[str, Dict[str, Set[str]]] = {}
        for src, by_type in self.adj.items():
            out[src] = {}
            for etype, targets in by_type.items():
                out[src][etype.name] = set(targets)
        return out

"""## auml_consistency/core/store.py"""

"""
core/store.py: UMLModel storage container.

This module defines the central container class :class:`UMLModel`, which
aggregates all model elements (via the ``elements`` dictionary) and the
associated dependency graph (:class:`DependencyGraph`). It provides a
single, canonical in-memory representation of the model used by all
subsystems (parsers, generators, validators).
"""
```

```
import logging
from typing import Dict, Optional
from auml_consistency.core.model import ModelElement
from auml_consistency.core.dependency_graph import DependencyGraph

# Setup logger
logger = logging.getLogger(__name__)

class UMLModel:
    """
    Central storage class representing a single internal UML model instance.

    The model maintains:
    - a mapping from element identifiers to :class:`ModelElement` instances;
    - a dependency graph that encodes structural, semantic and traceability
      relationships between elements.
    """
    def __init__(self):
        # Main storage of all elements keyed by their unique identifiers.
        self.elements: Dict[str, ModelElement] = {}
        # Associated dependency graph built over the elements of this model.
        self.dependency_graph: DependencyGraph = DependencyGraph()

    def add_element(self, element: ModelElement) -> None:
        """
        Add a model element to the storage.

        Raises
        ------
        ValueError
            If an element with the same identifier already exists. This guards
            against accidental ID collisions and ensures the uniqueness invariants.
        """
        if element.id in self.elements:
            existing = self.elements[element.id]
            msg = (f"Duplicate ID detected: {element.id}. "
                   f"New element '{element.name}' conflicts with existing
'{existing.name}'.")
            logger.error(msg)
            raise ValueError(msg)

        self.elements[element.id] = element

    def get_element_by_id(self, id: str) -> Optional[ModelElement]:
        """Return a model element by its unique identifier, if present."""
        return self.elements.get(id)

    def rebuild_graph(self):
        """
        Rebuild the dependency graph from the current set of elements.

        This method constructs a fresh :class:`DependencyGraph` instance and
        delegates the population of edges to
        :meth:`DependencyGraph.build_from_model`. The old graph is discarded
        after a successful rebuild (swap pattern).
        """
        logger.debug("Rebuilding dependency graph...")
        new_graph = DependencyGraph()
        new_graph.build_from_model(self)
        self.dependency_graph = new_graph
        logger.debug("Dependency graph rebuild complete.")

    def remove_element(self, element_id: str):
        """
```

```
        Remove an element from the model and update the dependency graph.

        The element is removed from the ``elements`` dictionary (if present),
        and the corresponding vertex and all incident edges are removed from
        the dependency graph, keeping :math:`G_M` consistent with the model.
        """
        if element_id in self.elements:
            el = self.elements[element_id]
            del self.elements[element_id]
            logger.info("Removed element %s (%s) from store", element_id, el.name)
        else:
            logger.warning("Attempted to remove non-existent element %s", element_id)

        self.dependency_graph.remove_element(element_id)

"""## auml_consistency/parsers/__init__.py

parsers package: source-to-model extraction.

This package contains parsers that translate concrete source artifacts
(for example, Python code) into the canonical UMLModel representation.
"

## auml_consistency/parsers/python_parser.py
"""

"""
python_parser.py: Python source code analyzer.

This module implements an AST-based parser for Python source code. It traverses
the abstract syntax tree (AST), extracts structural and behavioral information
(including selected docstring annotations), and populates a :class:`UMLModel`.
"""

from __future__ import annotations

import ast
import re
import logging
from pathlib import Path
from typing import List, Optional, Dict

from auml_consistency.core.model import (
    UMLClass,
    UMLOperation,
    UMLAttribute,
    UMLParameter,
    UMLLifeline,
    UMLMessage,
    UMLPackage,
    SourceLocation,
)
from auml_consistency.core.store import UMLModel
from auml_consistency.core.utils import generate_stable_id

# Setup logger
logger = logging.getLogger(__name__)

class ParsingError(Exception):
    """
    Domain-specific exception raised for structural parsing errors.

    This is used for violations of structural invariants such as σ₂
    (uniqueness of class names within a package).
```

```
    """
    pass

class ASTParserVisitor(ast.NodeVisitor):
    """
    AST visitor that extracts UML elements from Python source code.

    The visitor implements the classical :class:`ast.NodeVisitor` pattern and
    incrementally populates a shared :class:`UMLModel` instance while traversing
    the AST of a single file. It preserves a minimal context (current class and
    function) required for accurate extraction.
    """

    def __init__(self, model: UMLModel, file_path: str):
        """
        Initialize the visitor with a model instance and file namespace.

        Parameters
        ----------
        model:
            The :class:`UMLModel` instance to be populated.
        file_path:
            A relative path used as a namespace for stable identifier
            generation.
        """
        self.model = model
        self.file_path = file_path   # relative path (namespace) for stable IDs
        self.current_class_id: Optional[str] = None
        self.current_class: Optional[UMLClass] = None
        self.current_function_name: Optional[str] = None

    def _get_loc(self, node: ast.AST) -> SourceLocation:
        """Compute a :class:`SourceLocation` triple for the given AST node."""
        lineno = getattr(node, "lineno", 0)
        col = getattr(node, "col_offset", 0)
        return (self.file_path, lineno, col)

    def _generate_id(self, node_name: str, line: int) -> str:
        """
        Generate a stable identifier for an AST node.

        The identifier is derived from the namespace (relative file path),
        logical node name and line number via :func:`generate_stable_id`.
        It is used for classes and operations where the line number is
        meaningful for traceability.
        """
        return generate_stable_id(self.file_path, node_name, line)

    def _member_id(self, member_name: str) -> str:
        """
        Generate a stable identifier for a class member.

        The identifier is derived from the file namespace, class name and
        member name. It intentionally ignores the physical line number so
        that the same logical member retains its ID across refactorings
        that only change ordering or spacing.
        """
        cls_name = self.current_class.name if self.current_class else "<module>"
        # line = 0 by convention to achieve stability within a class.
        return generate_stable_id(self.file_path, f"{cls_name}.{member_name}", 0)

    # ---- Class nodes ----
    def visit_ClassDef(self, node: ast.ClassDef):
        """
```

```
    Process a ``class ...`` definition.

    The method extracts the class name, its base classes, creates a
    corresponding :class:`UMLClass` element, and sets the current class
    context for nested elements.
    """
    class_id = self._generate_id(node.name, getattr(node, "lineno", 0))
    logger.debug("Visiting class: %s (ID: %s)", node.name, class_id)

    # 1. Analyze base classes (inheritance).
    super_classes_ids: List[str] = []
    for base in node.bases:
        if isinstance(base, ast.Name):
            super_classes_ids.append(base.id)
        elif isinstance(base, ast.Attribute):
            # Reconstruct dotted name (e.g., module.Class).
            parts: List[str] = []
            cur = base
            while isinstance(cur, ast.Attribute):
                parts.append(cur.attr)
                cur = cur.value
            if isinstance(cur, ast.Name):
                parts.append(cur.id)
            dotted = ".".join(reversed(parts))
            super_classes_ids.append(dotted)
        else:
            # Fallback: use an unparsed string representation.
            try:
                super_classes_ids.append(ast.unparse(base))
            except Exception:
                super_classes_ids.append(str(base))

    # 2. Create UMLClass with empty attribute/operation lists.
    uml_class = UMLClass(
        id=class_id,
        name=node.name,
        source_location=self._get_loc(node),
        super_classes=super_classes_ids,
        attributes=[],
        operations=[],
    )

    self.model.add_element(uml_class)

    # 3. Establish context for nested elements.
    self.current_class_id = class_id
    self.current_class = uml_class

    # 4. Recursively visit the body of the class.
    self.generic_visit(node)

    # 5. Reset context.
    self.current_class_id = None
    self.current_class = None

def visit_FunctionDef(self, node: ast.FunctionDef):
    """
    Process a ``def ...`` definition (methods inside a class).

    Functions not belonging to a class are traversed but do not generate
    UML operations in the current prototype.
    """
    # Preserve the previous function context for nested visits.
    prev_function = self.current_function_name
```

```
self.current_function_name = node.name

if not self.current_class_id or not self.current_class:
    # Ignore functions outside classes (still traverse the subtree).
    self.generic_visit(node)
    self.current_function_name = prev_function
    return

op_id = self._generate_id(node.name, getattr(node, "lineno", 0))

# 1. Parse parameters.
params: List[UMLParameter] = []
for arg in node.args.args:
    if arg.arg == "self":
        continue

    param_type = "Any"
    if getattr(arg, "annotation", None):
        try:
            param_type = ast.unparse(arg.annotation)
        except Exception:
            param_type = "Any"

    params.append(
        UMLParameter(
            id=f"{op_id}_param_{arg.arg}",
            name=arg.arg,
            type=param_type,
        )
    )

# 2. Parse return type.
return_type = "Any"
if getattr(node, "returns", None):
    try:
        return_type = ast.unparse(node.returns)
    except Exception:
        return_type = "Any"

# 3. Create UMLOperation.
uml_op = UMLOperation(
    id=op_id,
    name=node.name,
    source_location=self._get_loc(node),
    parameters=params,
    return_type=return_type,
)

# 4. Analyze docstring (behavioral annotations).
docstring = ast.get_docstring(node)
if docstring:
    if "@hasBehavior" in docstring:
        uml_op.hasBehavior = True

    # Extract @sequence tags: collect all lines following @sequence.
    sequence_matches = re.findall(r"@sequence\s+(.*)", docstring, flags=re.M)
    sequence_lines: List[str] = []
    if sequence_matches:
        # sequence_matches may contain fragments; normalize into lines.
        for match in sequence_matches:
            for part in re.split(r"[\r\n]+", match):
                for line in part.split(";"):
                    line = line.strip()
                    if line:
```

```
                        sequence_lines.append(line)
            if sequence_lines:
                self._parse_sequence_docstring(sequence_lines, uml_op)

    # 5. Register operation in the model and attach to the parent class.
    self.model.add_element(uml_op)
    self.current_class.operations.append(uml_op)

    # Recursively traverse the body (e.g., to discover self.x in __init__).
    self.generic_visit(node)

    # Restore previous function context.
    self.current_function_name = prev_function
```