МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 56.00.00.000 ПЗ

Група ІПм-23-3

Репецький Андрій - Іван

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Репецький Андрій – Іван Олегович

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

МАГІСТЕРСЬКА РОБОТА

Засоби концептуальної схеми валідації на основі тестування в модельно-орієнтованому середовищі

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Репецький А.-І.О.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Михайлюк Ірина Романівна, к.п.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2024 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Репецькому Андрію – Івану Олеговичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Засоби концептуальної схеми валідації на основі тестування в модельно-орієнтованому середовищі ”

керівник проекту (роботи) Михайлюк Ірина Романівна, к.п.н., доцент

затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781/7

2. Строк подання студентом проекту (роботи) 15 грудня 2024 р.

3. Вихідні дані до проекту (роботи) Теоретичні концепції та формальні моделі побудови та функціонування інформаційних технологій модельно-орієнтованих середовищ

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Представлення області застосування концептуальної схеми валідації
2. Методи для покращення якості концептуальних схем
3. Методи та моделі валідації концептуальної схеми на основі процесів тестування
4. Імплементація моделей та засобів концептуальної схеми валідації

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Рівні вимог до комунікаційного аналізу та робочий процес (рис. 1.1)
2. Фрагмент метамоделі діаграми класів UML (рис. 1.2)
3. Фрагмент концептуальної схеми для Video Club на основі UML (рис. 1.3)
4. Приклад обмежень для системи Video Club (рис. 1.4)
5. Відносини між концептуальними сутностями (рис. 1.5)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц., к.т.н. Вовк Р.Б.	

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі магістерської роботи	15.09.2024	виконано
2	Аналіз концепцій та алгоритмів предметної області	29.09.2024	виконано
3	Представлення області застосування концептуальної схеми валідації	15.10.2024	виконано
4	Методи для покращення якості концептуальних схем	08.11.2024	виконано
5	Методи та моделі валідації концептуальної схеми на основі процесів тестування	20.11.2024	виконано
6	Імплементація моделей та засобів концептуальної схеми валідації	01.12.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 81 с., 27 рис., 4 табл., 55 джерел.

Тема: Засоби концептуальної схеми валідації на основі тестування в модельно-орієнтованому середовищі

Об'єкт дослідження: процеси моделювання та валідації концептуальних схем у розробці програмного забезпечення.

Мета роботи: імплементація моделей, методів і засобів валідації концептуальних схем, що забезпечують їхню відповідність вимогам, синтаксичну та семантичну правильність.

Предмет дослідження: методи, моделі та засоби валідації концептуальних схем на основі тестування в модельно-орієнтованому середовищі.

Результати дослідження

В роботі реалізовано виконувані концептуальні схеми з інтеграцією засобів автоматизованої валідації в модельно-орієнтоване середовище. Визначено критерії для оцінки якості концептуальних схем із урахуванням їхньої повноти, правильності та актуальності.

Висновок

Імплементація моделей та засобів валідації концептуальних схем у модельно-орієнтованому середовищі забезпечує більш точне, гнучке та ефективне тестування. Використання UML як основи дозволяє досягти високого рівня стандартизації та автоматизації, що сприяє створенню якісних програмних рішень.

КОНЦЕПТУАЛЬНА СХЕМА, UML, ВАЛІДАЦІЯ, ТЕСТУВАННЯ, МОДЕЛЬНО-ОРІЄНТОВАНЕ СЕРЕДОВИЩЕ, КОМУНІКАЦІЙНИЙ АНАЛІЗ, ГЕНЕРАЦІЯ ТЕСТІВ, ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ABSTRACT

Master Thesis: 81 pp., 27 fig., 4 tab., 55 sources.

Thesis Subject: Conceptual schema validation tools based on testing in a model-oriented environment

Object of the study: processes of modeling and validation of conceptual schemas in software development.

Purpose of the work: implementation of models, methods and tools for validating conceptual schemas that ensure their compliance with requirements, syntactic and semantic correctness.

Subject of the study: methods, models and tools for validating conceptual schemas based on testing in a model-oriented environment.

Research results

The work implements executable conceptual schemas with the integration of automated validation tools into a model-oriented environment. Criteria for assessing the quality of conceptual schemas are determined, taking into account their completeness, correctness and relevance.

Conclusion

Implementation of models and tools for validating conceptual schemas in a model-oriented environment provides more accurate, flexible and effective testing. Using UML as a basis allows achieving a high level of standardization and automation, which contributes to the creation of high-quality software solutions.

CONCEPTUAL DIAGRAM, UML, VALIDATION, TESTING, MODEL-ORIENTED ENVIRONMENT, COMMUNICATION ANALYSIS, TEST GENERATION, SOFTWARE QUALITY

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ALF - Activity Language Framework

CS – Conceptual Schema

CA - Communication Analysis

CIM - Computation-Independent Model

CED - Communicative Event Diagram

PIM - Platform-Independent Model

PSM - Platform-Specific Model

MDE – Model-Driven Engineering

MDD – Model-Driven Development

DSL – Domain-Specific Language

TDD – Test-Driven Development

BDD – Behavior-Driven Development

CI/CD – Continuous Integration/Continuous Deployment

SUT – System Under Test

VCS – Version Control System

COTS – Commercial Off-The-Shelf

FMEA – Failure Mode and Effects Analysis

OMG – Object Management Group

OCL – Object Constraint Language

ORM – Object-Relational Mapping

DFA – Deterministic Finite Automaton

CRS – Change Request System

MoM – Model of Models

RTM – Requirement Traceability Matrix

TOGAF – The Open Group Architecture Framework

ВСТУП

Актуальність теми.

В умовах стрімкого розвитку інформаційних технологій і зростання складності програмного забезпечення забезпечення якості стає ключовим завданням у процесах його розробки. Помилки на ранніх етапах проєктування програмного забезпечення, зокрема в концептуальних схемах, можуть спричинити суттєві проблеми на етапах розгортання та експлуатації, збільшення витрат і зниження задоволеності кінцевих користувачів.

Концептуальні схеми є основою для формалізації вимог, проєктування архітектури та побудови взаємозв'язків між компонентами системи. Проте їх ефективність безпосередньо залежить від якості, що визначається такими параметрами, як повнота, правильність, актуальність і семантична відповідність. Відсутність ефективних методів валідації концептуальних схем може призвести до того, що дефекти залишаться непоміченими, що ускладнює процес розробки та створює ризик для функціонування системи.

Особливу увагу привертає використання модельно-орієнтованого підходу (MDA), який дозволяє застосовувати стандартизовані методи моделювання, такі як UML, для створення концептуальних схем. Водночас зростає попит на автоматизацію перевірки якості цих схем за допомогою тестування, що дозволяє суттєво скоротити час і витрати на валідацію.

Актуальність дослідження також обумовлена потребою у створенні динамічно виконуваних концептуальних схем, які можуть бути інтегровані у середовище розробки для реального часу перевірки та аналізу. Це сприяє ранньому виявленню дефектів, зменшує ризики та підвищує ефективність роботи команди розробників.

З огляду на вищезазначене, розробка моделей, методів і засобів валідації концептуальних схем у модельно-орієнтованому середовищі є важливим науковим і практичним завданням, що сприяє підвищенню якості

програмного забезпечення, оптимізації процесів його розробки та відповідності зростаючим вимогам до сучасних інформаційних систем.

Мета дослідження - імплементація моделей, методів і засобів валідації концептуальних схем, що забезпечують їхню відповідність вимогам, синтаксичну та семантичну правильність.

Об'єкт дослідження - процеси моделювання та валідації концептуальних схем у розробці програмного забезпечення.

Предмет дослідження - методи, моделі та засоби валідації концептуальних схем на основі тестування в модельно-орієнтованому середовищі.

Задачі дослідження:

- Дослідити роль концептуальних схем у процесах верифікації та валідації програмного забезпечення.

- Визначити критерії якості концептуальних схем та розробити підхід до їхньої оцінки.

- Розробити методологію валідації концептуальних схем із застосуванням процесів тестування.

- Імплементувати моделі та засоби валідації в модельно-орієнтоване середовище.

Методи дослідження

- Аналітичні методи для вивчення вимог до концептуальних схем.
- Методи комунікаційного аналізу для моделювання вимог.
- Формальні методи перевірки синтаксису та семантики UML-схем.
- Автоматизовані підходи до тестування та генерації тестів.
- Методики оцінки якості концептуальних схем.

Наукова новизна отриманих результатів

Розроблено комплексну методологію валідації концептуальних схем, що інтегрує етапи аналізу вимог, тестування та оцінки якості, запропоновано підходи до генерації тестів для динамічних UML-схем.

Практичне значення результатів

Результати дослідження можуть бути використані в розробці програмного забезпечення для підвищення якості моделювання на ранніх етапах життєвого циклу ПЗ та автоматизації процесів тестування концептуальних схем.

Структура магістерської роботи. Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 81 сторінку, і містить 27 рисунків, 5 таблиць, список використаних джерел із 55 найменувань.

РОЗДІЛ 1. ПРЕДСТАВЛЕННЯ ОБЛАСТІ ЗАСТОСУВАННЯ КОНЦЕПТУАЛЬНОЇ СХЕМИ ВАЛІДАЦІЇ В ПРОЦЕСІ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Поняття концептуальної схеми в процесах верифікації та валідації

Широкий спектр методів розробки програмного забезпечення підтримує розробку інформаційних систем (ІС), розглядаючи розробку вимог як важливу діяльність, яка визначає загальні знання про область ІС та функції, які вона повинна виконувати. У галузі інформаційних систем це знання називається концептуальною схемою [1].

Згідно з [2], концептуальна схема або концептуальна модель — це “високорівневий опис програми. У ній перераховуються всі концепції програми, з якими можуть зіткнутися користувачі, описується, як ці концепції співвідносяться одна з одною, і як ці концепції вписуються в завдання, які користувачі виконують за допомогою програми” .

У розробці, керованій моделлю, основними артефактами є концептуальні схеми (CS - conceptual schema) або моделі, і зусилля зосереджені на їх створенні, тестуванні та еволюції на різних рівнях абстракції через перетворення. Якщо концептуальна схема має дефекти, вони передаються на наступні етапи, включаючи кодування. Тому для забезпечення правильної генерації кінцевих програмних продуктів необхідно впроваджувати методи підвищення якості концептуальних схем . Одним із завдань розробки, керованої моделлю, є можливість генерувати тестові приклади на основі вимог, а не лише для виявлення дефектів, а також для ранньої перевірки вимог на рівні концептуальних схем, щоб можна було прийняти відповідні рішення. на основі результатів процесу перевірки, щоб допомогти зменшити витрати на розробку та покращити якість програмного

забезпечення. У цій роботі ми розробили підхід до перевірки концептуальної схеми на основі тестування з метою покращення якості.

Незважаючи на великий скептицизм і численні проблеми [3], розроблення, кероване моделлю (MDD), використовується та вдосконалюється, щоб забезпечити багаточисельні потенційні переваги для промисловості [4 - 5]. Однією з його найбільших переваг є здатність впоратися зі складністю розробки програмного забезпечення шляхом підвищення рівня абстракції.

Моделі виражаються за допомогою концепцій, які не пов'язані з конкретною технологією реалізації (наприклад, уніфікована мова моделювання UML, мова обмежень об'єктів OCL, мова дій для фундаментального UML - ALF), що означає, що моделі можна легше визначити, зрозуміти, підтримувати та документувати. Як і в інженерії, керованій моделями (MDE), основними артефактами є концептуальні моделі, і забезпечення їхньої якості на оптимальному рівні все ще є проблемою для дослідників і розробників.

Хоча верифікація та валідація (V&V) тісно пов'язані з концепціями якості та забезпечення якості програмного забезпечення, дуже небагато інструментів MDD включають ці дії в процес розробки. OO-Method (OOM) [6], підхід на основі моделі, керованої архітектурою (MDA), є ініціативою, керованою моделлю, з технічною мультипрограмою (структурна модель, динамічна модель, функціональна модель і презентаційна модель), де структурний вигляд є основою для автоматичного виведення інших представлень, і ця функція допомагає мінімізувати такі проблеми, як специфіка кількох зображень і синхронізація, інтеграція та поширення змін. ОО-метод був успішно впроваджений у промисловості за допомогою комерційного інструменту Integranova (раніше відомого як OLIVANOVA). Цей інструмент керує синтаксичною перевіркою концептуальних схем (наприклад, синтаксичної правильності) [6], але він все ще не підтверджує, чи відповідає побудована модель вимогам і очікуванням зацікавлених сторін.

З постійно зростаючою складністю програмних систем здатність ідентифікувати переважну більшість дефектів на ранніх етапах на рівні моделі є проблемою, яка, якщо її вирішити, може допомогти зменшити витрати на розробку та покращити якість програмного забезпечення [7]. Список відкритих проблем, представлений у [8], включає повні та правильні концептуальні схеми.

Однак, щоб оцінити якість концептуальної схеми, нам потрібна модель якості. У літературі ми можемо знайти кілька пропозицій, наприклад [9, 10]. Хоча в [11] припускають, що необхідно більше працювати над оцінкою якості моделі. Ми прагнемо встановити властивості якості, які можна покращити за допомогою методів тестування.

Тестування є частиною процесу V&V, де концептуальна схема працює в контрольованих умовах, (1) щоб перевірити, що вона поводиться як зазначено; (2) для виявлення дефектів і (3) для підтвердження вимог користувача [12]. Таким чином, тісна інтеграція між моделлю та кодом у розробці, керованій моделлю, розробка мов високого рівня, придатних для моделювання CS (наприклад, UML/OCL з мовою ALF), породжує потребу в розробці перевірки і стратегії перевірки, які слід застосовувати на ранніх стадіях життєвого циклу програмного забезпечення (наприклад, на рівні CS), а також для виявлення та виявлення дефектів у реалістичних схемах з мінімальними витратами.

Ця робота має на меті визначити базу перевірки на основі тестування для багаторакурсних концептуальних схем (тобто структурних і поведінкових). Ми зосередимося на адаптації методів тестування для середовищ, керованих моделлю, таких як підхід ОО-метод, тому що ми вважаємо, що тестування може бути дуже ефективним і ефективним способом виявлення дефектів на ранній стадії та може відігравати важливу роль у перевірці валідації концептуальні схеми.

Помилки вимог є найчастішою причиною дефектів у проектах розробки системи [13]. Це свідчить про те, що було б більш ефективним і

результативним зосередити зусилля із забезпечення якості на ранніх стадіях, щоб виявляти дефекти, як тільки вони виникають. У MDD здатність виявляти дефекти на ранній стадії все ще є проблемою, яка, якщо її вирішити, може допомогти зменшити витрати на розробку та підвищити якість програмних систем, що поставляються [7, 8]. Повинні бути реалізовані легкі методи тестування для покращення якості концептуальних схем. Ці методи повинні бути в змозі знайти дефекти з мінімальними зусиллями та без потреби в сильному тестуванні.

1.2. Концептуальні моделі як основа процесу моделювання та розробки програмного забезпечення

У контексті MDD, де концептуальні схеми (моделі) є основою всього процесу розробки, якість CS має великий вплив на кінцеву якість програмних систем, отриманих з них [20]. Отже, CS можуть безпосередньо впливати як на ефективність (час, вартість, зусилля), так і на ефективність (якість результатів) розробки інформаційних систем.

Концептуальні схеми розробляються за допомогою мови моделювання. Стандартом де-факто для аналізу та проектування об’єктно-орієнтованих програмних систем є Уніфікована мова моделювання (UML) [16], яка розширена обмеженнями OCL (мова обмежень об’єктів) [17]. Різноманітність діаграм UML забезпечує гнучкість і застосовність для розробників моделей для створення CS в різних просторах, де вони можуть бути використані (проблема, рішення та фон) [18]. Однак, оскільки процес моделювання є людським завданням, важко уникнути внесення дефектів у CS (наприклад, невідповідність, неправильні, зайві та неточні елементи).

Хоча дефекти можуть бути неминучими, ми повинні мінімізувати їх кількість і вплив на якість програмного забезпечення шляхом тестування та/або перевірки CS. Тестування спрямоване на виявлення дефектів у системі шляхом порівняння очікуваних результатів (виражених у системних вимогах)

із спостережуваними результатами (поведінка впровадження тестованої системи (SUT). У багатьох організаціях процеси тестування починаються після завершення коду. [19]. Щоб виявляти дефекти до того, як їх виправлення стане надзвичайно дорогим, і керувати неминучими змінами протягом життєвого циклу програмного забезпечення, тестування має розпочатися якнайшвидше (на рівні вимог) у життєвому циклі програмного забезпечення та інформації про типи дефектів, які виникають на ранніх етапах життєвого циклу розробки програмного забезпечення, можна використовувати для надання зворотного зв'язку зацікавленим сторонам (наприклад, розробникам моделей, тестувальникам) щодо виявлення дефектів і способів їх відстеження, зменшення та усунення, якщо мета полягає в тому, щоб отримати CS хорошої якості, інформація про кожен дефект має бути пов'язана з цілями якості, на які це впливає, відповідно до відповідної моделі якості для моделей у контексті MDD, як запропоновано в [9].

Для визначення вимог у документі системних вимог можна використати кілька методів. Найпопулярнішими методами визначення вимог користувача є природна мова та випадки використання. Незважаючи на те, що природна мова має кілька недоліків, таких як двозначність, нечіткість і надмірності, це найбільш часто використовувана техніка для опису вимог у промисловості. З іншого боку, варіанти використання зосереджені на більш структурованому описі взаємодії між різними користувачами та системою за допомогою графічного та текстового представлення, однак вони також мають форму природної мови та також мають вищезазначені недоліки.

1.2.1. Моделювання вимог на основі комунікаційного аналізу

Комунікаційний аналіз – це метод розробки вимог, який аналізує комунікативну взаємодію між інформаційними системами компанії (CIS) та їх середовищем [21].

Методологічною основою комунікаційного аналізу є етап аналізу інформаційної системи, результатом якого є специфікація аналізу, комунікаційно-орієнтована документація, яка описує інформаційну систему. Для цього CA (Communication Analysis) пропонує структуру вимог із п’ятьма рівнями (рис. 1.1):

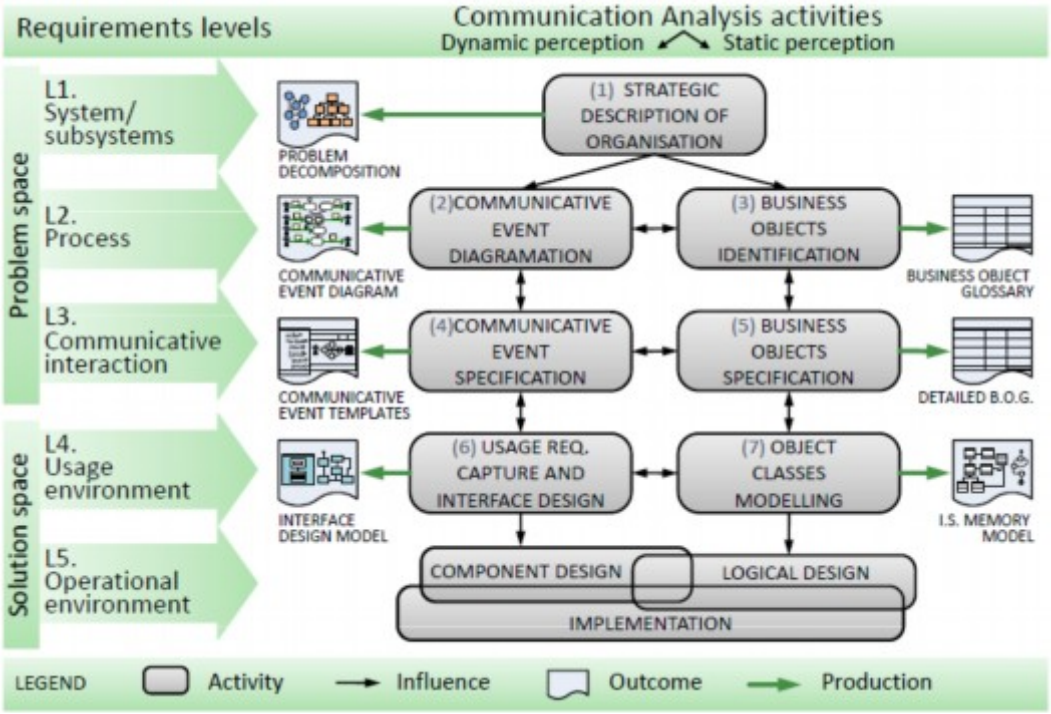

Рис. 1.1. Рівні вимог до комунікаційного аналізу та робочий процес

1) Рівень системи/підсистем (L1) відноситься до загального опису організації та її середовища (організаційна система та суб’єктна система відповідно), а також передбачає декомпозицію проблеми з метою зменшення її складності,

2) Рівень процесу (L2) відноситься до опису бізнес-процесу як з динамічної точки зору (шляхом ідентифікації потоків комунікативних взаємодій, так само як комунікативні події), так і зі статичної точки зору (шляхом ідентифікації бізнес-об’єктів),

3) Рівень комунікативної взаємодії (L3) стосується детального опису кожної комунікативної події (наприклад, опис пов’язаного з нею повідомлення) і кожного бізнес-об’єкта,

4) Рівень середовища використання (L4) стосується фіксації вимог, пов’язаних із використанням комп’ютерна інформаційна система (CIS), проектування інтерфейсів користувача та моделювання класів об’єктів, які підтримуватимуть пам’ять IS,

5) Рівень операційного середовища (L5) відноситься до розробки та реалізації програмного забезпечення CIS компоненти та архітектура [21].

1.2.2. Концепції якості концептуальної схеми

Значення якості широко обговорюється, і всі погоджуються, що якість є важливою властивістю продукції. ISO/IEC 9126 [22] (міжнародний стандарт для оцінювання якості програмного забезпечення, що відповідає ISO 9000 [23], група стандартів, пов’язаних з управлінням якістю) визначає якість програмного забезпечення як: «Сукупність функцій і характеристик продукту чи послуги, які впливають на його здатність задовольняти заявлені чи неявні потреби». Тому більшість підходів до оцінки якості розкладають концепцію якості на набір властивостей якості нижчого рівня (також званих «цілями», «атрибутами» або характеристиками якості), які можна точно виміряти.

У контексті моделювання якість CS або моделі — це ступінь присутності набору властивостей якості моделі. Таким чином, набір цілей якості з їхніми зв’язками, що супроводжується набором практик або засобів для досягнення цілей якості та методів оцінки для оцінки цілей якості, визначають модель якості [9].

1.3. Методи для покращення якості концептуальних схем

Щоб оцінити, чи відповідає CS зазначеним вище цілям якості, можна застосувати кілька методів. Усі ці методи спрямовані на валідацію та перевірку (V&V) CS відповідно до моделі якості.

IEEE [24] визначає валідацію як «підтвердження перевіркою та наданням об’єктивних доказів того, що конкретні вимоги для конкретного

передбачуваного використання виконані». З іншого боку, той самий стандарт визначає верифікацію як «підтвердження перевіркою та наданням об’єктивних доказів того, що визначені вимоги виконано».

Застосовуючи наведені вище визначення в контексті моделювання, валідація - це діяльність, яка відповідає на запитання: «Чи ми розробляємо правильну модель?», тобто чи всі знання в моделі є достатньо правильними та відповідними проблемній області. З іншого боку, верифікація — це діяльність, яка відповідає на питання «Чи правильно ми розробляємо модель?», тобто чи задовольняє модель такі властивості якості, як узгодженість. Відповідно до класифікації ISO/IEC 9126 валідація спрямована на перевірку зовнішньої якості, а верифікація — на перевірку внутрішньої якості.

Внески цієї роботи спрямовані на покращення перевірки концептуальної схеми в контексті розробки, керованої моделлю. Застосування методів, спрямованих на підтвердження вимог, може залежати від рівня формалізації специфікацій вимог.

Методи, що застосовуються до валідації, придатні для валідації програмного забезпечення в цілому та для валідації моделей (наприклад, моделі вимог, моделі дизайну, тестова модель тощо).

У цій роботі ми класифікуємо методи з двох точок зору:

1) спосіб виконання аналізу;

2) рівень формалізації.

Ця класифікація є надмірним спрощенням для цілей цієї роботи.

По-перше, щодо способу виконання аналізу ми класифікуємо методи на дві категорії:

Статичні методи. Статичні методи досліджують модель і міркують над усіма можливими поведінками, які можуть виникнути під час виконання [26]. Це означає, що модель зчитується людьми або переслідується комп’ютером, але не виконується як програма. Отже, статичні методи працюють під час «компіляції».

Динамічні методи. Динамічні методи працюють шляхом виконання програми (у нашому випадку CS або моделі) і спостереження за її виконанням [27].

Це означає, що модель запускається (або виконується) за допомогою комп'ютера. Отже, динамічні методи працюють під час виконання.

По-друге, щодо рівня формалізації ми класифікуємо методи на дві категорії:

Формальні методи. В [28] описують формальні методи як «математично засновані методи для опису властивостей системи. Такі формальні методи забезпечують структуру, в межах якої люди можуть специфікувати, розробляти, перевіряти та перевіряти системи систематично, а не випадково».

Неформальні методи. На відміну від формальних методів, неформальні методи не намагаються дотримуватися строгого підходу, а використовують неформальні методи. Перевага неформальних методів полягає в тому, що користувачеві не потрібно бути експертом у розумінні математичних моделей. Їх легко проілюструвати, і їх можна використовувати для перевірки моделей, написаних природною мовою, збільшуючи участь нетехнічних зацікавлених сторін. Як недолік, враховуючи їхню неформальність, вони можуть бути неоднозначними та надавати неточний результат.

Таблиця 1.1.

Відповідні методи валідації для концептуальних схем

	Static Methods	Dynamic Methods
Non-formal methods	Reviews Inspections	
Formal methods		Testing Simulation and Animation

Стандарт IEEE [29] визначає інспекцію як «візуальну перевірку програмного продукту для виявлення та виявлення аномалій програмного забезпечення, включаючи помилки та відхилення від стандартів і

специфікацій. Інспекції — це взаємні перевірки, які проводяться неупередженими фасилітаторами, навченими методам перевірки. виправлення або розслідування аномалії є обов’язковим елементом інспекції програмного забезпечення, хоча рішення не повинно бути визначено на інспекційній зустрічі». Порівняно з технічними оглядами та покроковими інструкціями, перевірки більш структуровані. Стандарт IEEE [29] стверджує, що перевірки повинні проводитися відповідно до плану проекту.

1.4. Концепції середовища, керованого моделлю

Як згадувалося раніше, це дослідження зосереджено на розробці системи перевірки на основі тестування, яка задовольняє кероване моделлю середовище з метою покращення якості концептуальної схеми. Цей розділ надає читачеві лексикон та інструменти, які використовуються під час тестування на основі моделі. По-перше, ми представляємо визначення та припущення MDA, а також концепції архітектури метамоделювання, які використовуються в нашій роботі. Потім ми підсумовуємо концепції концептуальних схем на основі компакт-дисків UML, а також виконуваного UML CS. Нарешті, ми підсумовуємо концепції стандартів OMG для специфікації виконуваних моделей.

1.4.1. Визначення та припущення MDA

Група управління об'єктами (OMG) визначила власну пропозицію щодо застосування практики MDE до розробки системи, яка називається MDA (Model-Driven Architecture). Вся інфраструктура MDA базується на кількох основних визначеннях і припущеннях. Основними елементами інтересу для MDA є наступні [39]:

- Система є предметом будь-якої специфікації MDA. Це може бути програма, окрема комп’ютерна система, деяка комбінація частин різних систем або об’єднання систем.

- Проблемний простір (або домен) — це контекст, у якому працює система.

- Простір рішень – це спектр можливих рішень, що задовольняють вимогам системи.

- Архітектура — це специфікація частин і з'єднувачів системи та правила взаємодії частин, які використовують з'єднувачі.

- Платформа — це набір підсистем і технологій, які забезпечують узгоджений набір функціональних можливостей, орієнтованих на досягнення визначеної мети.

- Метамодель є визначенням мови моделювання, яка забезпечує спосіб опису цілого класу моделей, які можуть бути представлені цією мовою. Таким чином, ми можемо визначити моделі реальності, а потім моделі, які описують моделі (так звані метамоделі), і рекурсивні моделі, які описують метамоделі (які називаються мета-метамоделями). Потім модель відповідає метамоделі так, як комп'ютерна програма відповідає граматиці мови програмування, на якій вона написана [39].

1.4.2. Огляд архітектури метамоделювання

Оскільки наша пропозиція відповідає принципам архітектури, керованої моделлю, вона розрізняє різні типи моделей на різних рівнях абстракції таким чином [39]:

Модель, незалежна від обчислень (CIM - Computation-Independent Model) — це найбільш абстрактний рівень моделювання, який представляє вимоги рішення без будь-яких прив'язок до обчислювальних наслідків.

Платформено-незалежна модель (PIM - Platform-Independent Model) — це рівень, який описує поведінку та структуру системи, незалежно від платформи реалізації.

Platform-Specific Model (PSM - Platform-Specific Model) містить всю необхідну інформацію щодо поведінки та структури програми на певній

платформі, яку розробники можуть використовувати для реалізації виконуваного коду.

Набір відображень між кожним рівнем і наступним може бути визначений за допомогою перетворень моделі. Як правило, кожен CIM може зіставлятися з різними PIM, які, у свою чергу, можуть зіставлятися з різними PSM.

У цій роботі діаграми класів UML (Unified Model Language) [40] визначають метамоделі, тоді як ATL (ATLAS - Transformation Language) [41] визначає перетворення моделі. Ми вибираємо обидва рішення, оскільки це добре відомі мови. UML запропонований OMG (Object Management Group) і часто використовується в MDE для визначення метамоделей. ATL є однією з найпопулярніших і широко використовуваних мов перетворення моделі [41]. ATL — це гібридна мова перетворення, яка містить суміш декларативних і імперативних конструкцій. Допоміжні та правила перетворення — це конструкції, які використовуються для визначення функціональності перетворення.

1.4.3. Концептуальні схеми на основі UML

Метою цієї роботи є розробка тестових випадків для пошуку недоліків у концептуальній схемі під час аналізу та проектування програмного забезпечення шляхом навмисної зміни CS на основі компакт-диска UML, що призводить до неправильної поведінки та, можливо, до збою. CS системи має описувати її структуру та поведінку (обмеження). У цій роботі для представлення такої CS використовується діаграма класів на основі UML.

Діаграма класів (див. рис. 1.2) є основним блоком UML, який показує елементи системи на абстрактному рівні (наприклад, клас, клас асоціації), їхні властивості (ownedAttribute), відносини (наприклад, асоціація та узагальнення) та операції.

В UML операція визначається шляхом визначення попередніх і післяумов. Нижче, на рисунку 1.2 показано фрагмент структури UML для

діаграми класів і виділено вісім елементів, що представляють інтерес для цієї роботи.

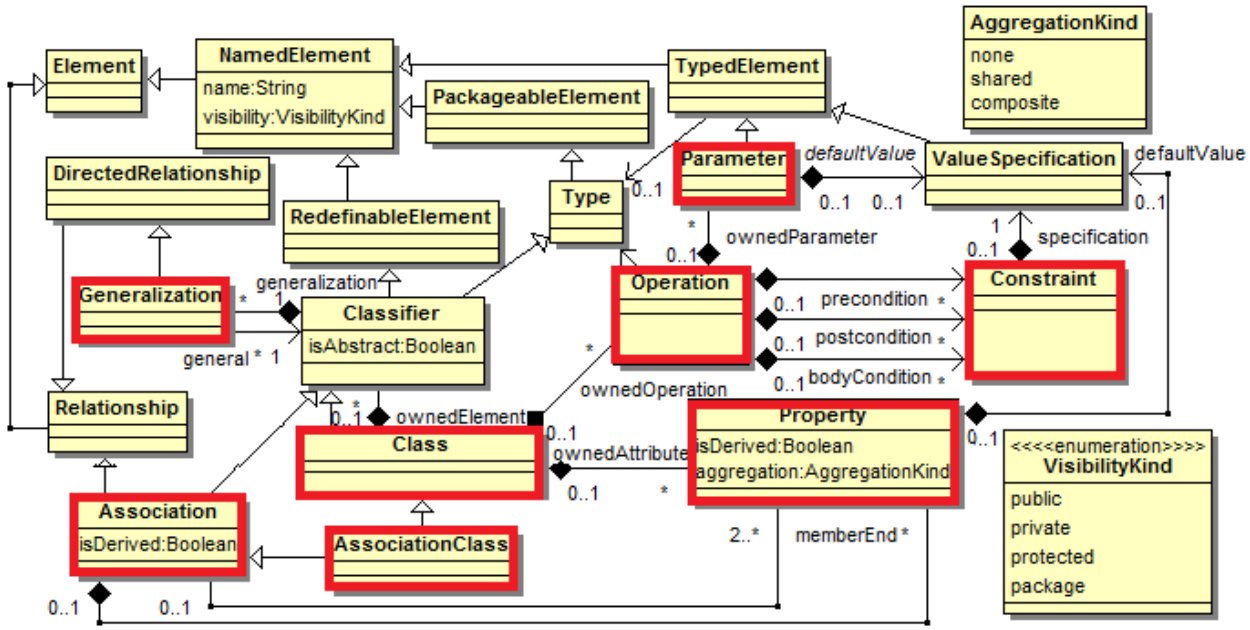

Рис. 1.2. Фрагмент метамоделі діаграми класів UML

1.4.4. Динамічно виконувана концептуальна схема UML

Якщо ми хочемо динамічно тестувати моделі для виявлення потенційних неправильних уявлень, виражених у них, нам потрібно мати можливість виконувати моделі. Виконувана модель — це модель зі структурою (що це таке?) і специфікацією поведінки (що вона робить?), достатньо детальною для систематичного виконання у виробничому середовищі.

Структурна модель визначає статичну частину інформаційної системи [1], яка утворена набором класів, набором атрибутів кожного класу, набором асоціацій між класами, набором узагальнень серед класів і набором цілісності. обмеження (тобто умови, які повинні бути задоволені в усіх станах інформаційної системи).

Усі елементи в діаграмі класів вважаються коректними екземплярами відповідних метакласів метамоделі UML [40].

Деякі обмеження цілісності (в основному потужності) можуть бути представлені графічно в CD, тоді як інші можуть бути текстово визначені в OCL [17]. На рисунку 1.3 наведено фрагмент структурної моделі нашого відеоклубу CS.

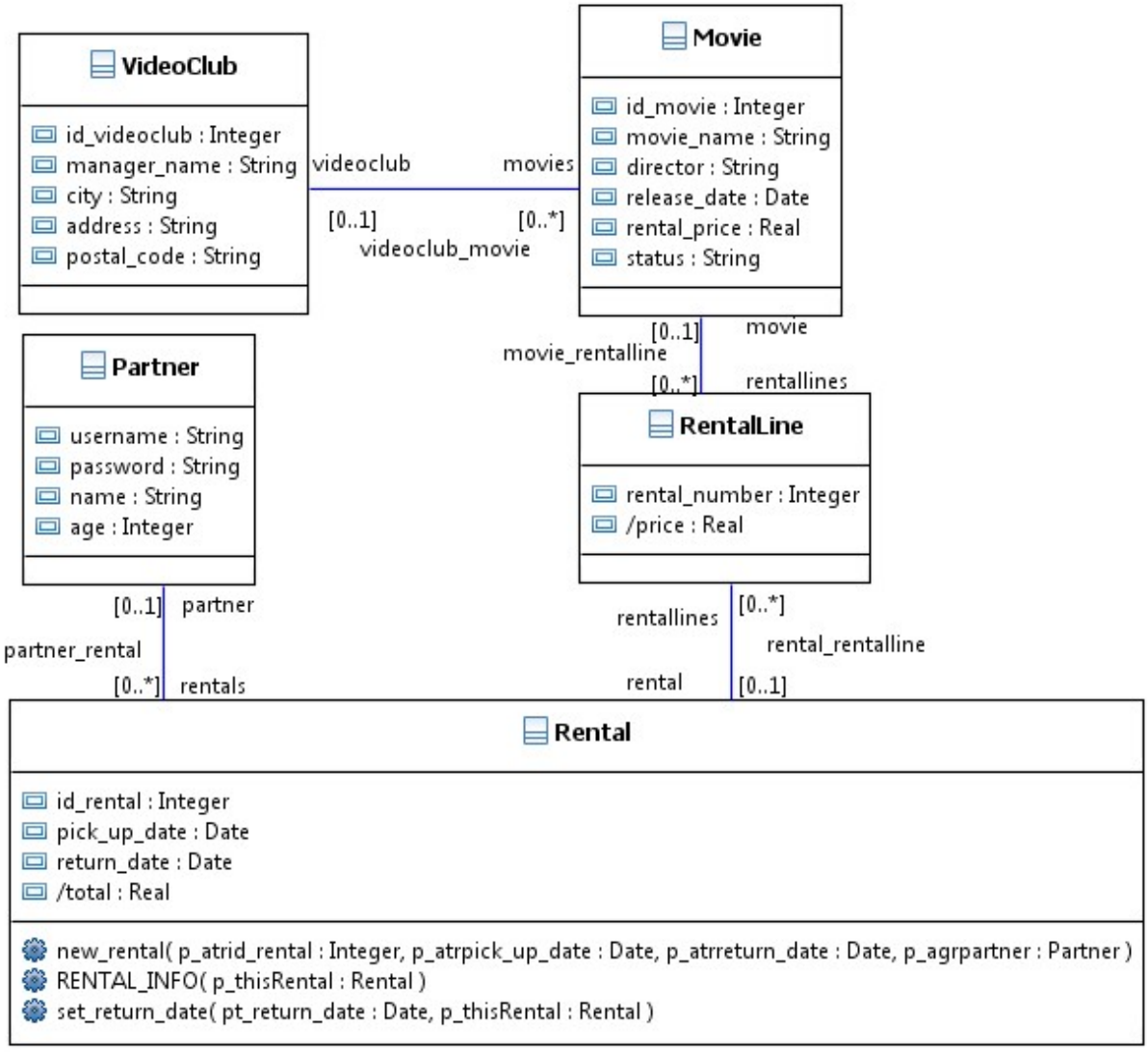

Рис. 1.3. Фрагмент концептуальної схеми для Video Club на основі UML

Поведінкова модель визначає динамічну частину інформаційної системи, тобто дійсні зміни в стані системи, а також функції, які система може виконувати [1]. В UML є кілька моделей для визначення поведінки системи на високому рівні абстракції, наприклад, за допомогою діаграм варіантів використання, діаграм активності, діаграм станів тощо. Однак, як

ми запровадили, щоб бути виконуваним, поведінкові моделі повинні бути достатньо детальними. З цієї причини в цій дисертації, щоб визначити детальну модель поведінки, ми використовуємо операції. Операції — це послідовності атомарних кроків, які користувачі можуть виконувати, щоб запитувати та/або змінювати інформацію, змодельовану в структурній моделі. Операції додаються до класів UML.

На рисунку 1.3 показано три операції, пов'язані з класом Rental (тобто нова оренда, RENTAL_INFO та set_return_date).

Крім того, попередні та післяумови та інваріанти, включені до діаграми класів, також є операціями або частиною операцій (тобто попередні та післяумови). Наприклад, на рисунку 1.4 показано деякі обмеження, додані до класу Rental CS Video Club.

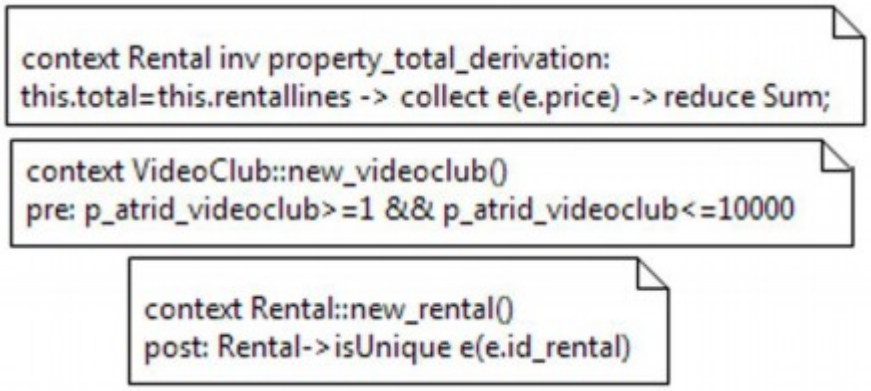

Рис. 1.4. Приклад обмежень для системи Video Club

1.4.5. Типи дефектів у концептуальних схемах на основі UML

Концептуальна схема не завжди може представляти функціональність, для якої вона призначена. Причини та наслідки відхилень від очікуваної функції в концептуальній схемі є факторами, які впливають на надійність і якість програмного продукту.

Коли дефект виявляється під час виконання моделі, він називається несправністю, але не є несправністю, якщо він виявлений під час перевірки або статичного аналізу. Таким чином, несправність є підтипом дефекту і

може спричинити збій, коли вона зустрічається. Ми скоригували опис обсягу зв'язків між концептуальними сутностями, запропонованими стандартом IEEE, з одного боку, з концептуальними сутностями нашого дослідження (концептуальними моделями на основі UML), з іншого. Це призвело до рисунку 1.5, де ці зв'язки зображені графічно. Червона рамка безпосередньо відповідає стандарту IEEE.

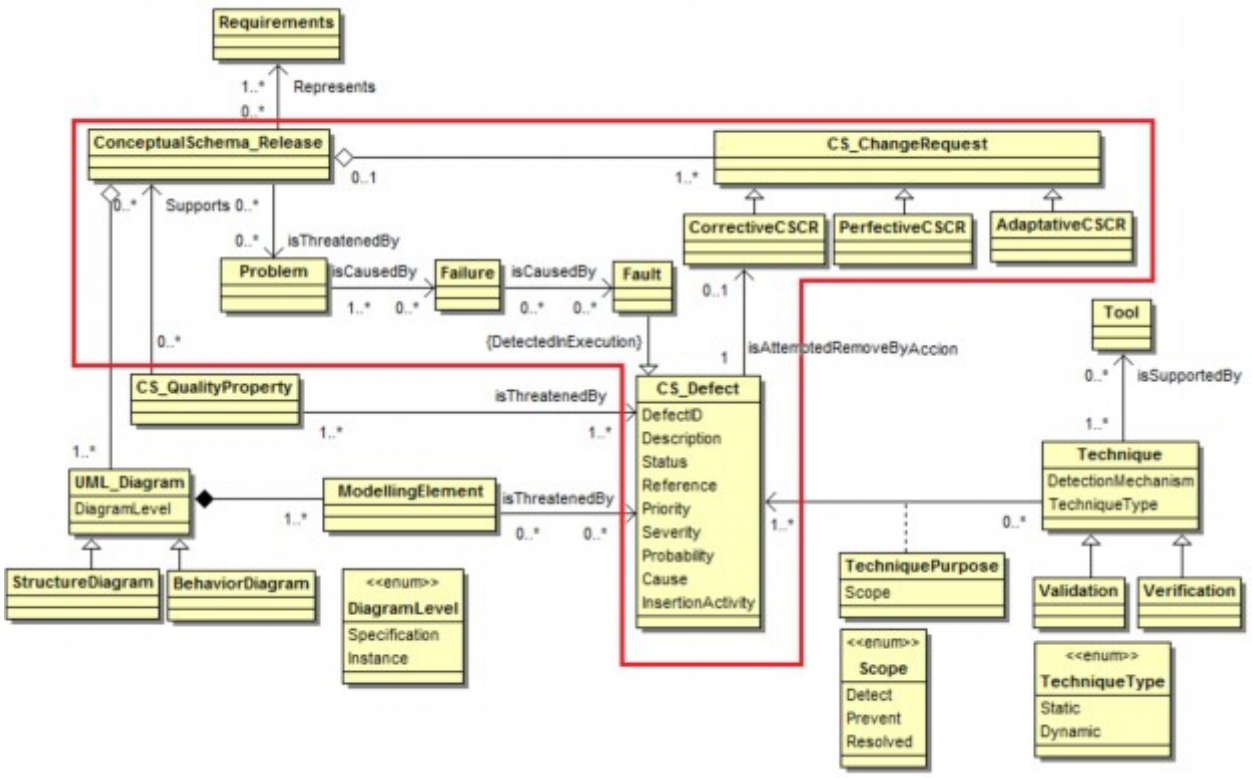

Рис. 1.5. Відносини між концептуальними сутностями

Як показано на рисунку 1.5, концептуальна схема представляє системні вимоги (програмного забезпечення) на абстрактному рівні. Вона може складатися з кількох діаграм UML (структурних і поведінкових), де кожен тип діаграми містить різний тип (елемент моделювання) інформації про систему.

Крім того, концептуальна схема має пов'язані властивості якості, які підтримують представлення або опис вимог. Цим властивостям якості зазвичай загрожують дефекти, які виникають на рівні елементів діаграми концептуальної схеми.

Дефект може бути пов’язаний з одним запитом на коригувальні зміни концептуальної схеми, який намагається усунути дефект, і кожен запит на коригувальні зміни може бути пов’язаний щонайбільше з одним випуском концептуальної схеми.

На рисунку 1.5 також показані дві інші причини запиту на зміну концептуальної схеми (CSCR): запит на ідеальну зміну концептуальної схеми та запит на адаптивну зміну концептуальної схеми.

Дефекти на концептуальному рівні можна знайти кількома способами за допомогою методів V&V, які використовують механізм виявлення (на основі правил, показників і умов моделювання) для цієї мети.

Відповідно до природи техніки, це може підтримуватися статично або динамічно інструментом і може мати тип області дії, який залежить від його призначення (тобто виявлення, запобігання та вирішення).

Дефекти мають вставну активність, серйозність, пріоритет і ймовірність виникнення. Вони виявляються в будь-який конкретний час, помічаючи конкретний опис (симптом) за допомогою механізму виявлення. Кожен із цих аспектів має значення для цілей необхідного аналізу, а також дозволяє класифікувати дефекти.

Висновки до розділу

Перший розділ присвячено розгляду концептуальних схем як фундаменту процесу валідації в розробці програмного забезпечення. У рамках дослідження були представлені ключові аспекти, що визначають значення концептуальних моделей та їх роль у забезпеченні якості розробки.

Концептуальні схеми є критично важливими в процесах верифікації та валідації, оскільки вони забезпечують формалізацію вимог, що сприяє точності та узгодженості програмного продукту. У розділі було розглянуто, як ці схеми формують основу для взаєморозуміння між замовником і розробниками, мінімізуючи ризик неправильного тлумачення вимог.

Особливу увагу приділено ролі концептуальних моделей у процесі моделювання та розробки програмного забезпечення. Моделювання вимог на основі комунікаційного аналізу дозволяє більш ефективно виявляти ключові потреби користувачів. Розглянуто підходи до визначення якості концептуальних схем, які впливають на подальшу розробку.

Різноманітні методи, спрямовані на підвищення якості концептуальних схем, забезпечують більш чітке уявлення про структуру системи, зменшують імовірність появи дефектів та підвищують точність проектування. Це включає підходи до перевірки внутрішньої консистентності схем та зменшення їх складності.

У межах MDA (Model-Driven Architecture) обговорюються припущення та особливості, які лежать в основі цього підходу. Розглянуто архітектуру метамоделювання, а також значення UML як стандарту для створення концептуальних схем. Особливий інтерес викликають динамічно виконувані UML-схеми, які дозволяють перевірити їх працездатність у реальному часі.

Важливим аспектом є класифікація типів дефектів у концептуальних схемах на основі UML. Це допомагає фокусуватися на проблемних аспектах моделювання та розробляти ефективні методи для їх усунення.

У підсумку, аналіз концептуальних схем демонструє, що вони є критичним інструментом для забезпечення якості програмного забезпечення. Систематичний підхід до їх створення, перевірки та вдосконалення сприяє створенню надійних і відповідних вимогам продуктів.

РОЗДІЛ 2. МЕТОДИ ТА МОДЕЛІ ВАЛІДАЦІЇ КОНЦЕПТУАЛЬНОЇ СХЕМИ НА ОСНОВІ ПРОЦЕСІВ ТЕСТУВАННЯ

2.1. Особливості перевірки концептуальних схем програмного забезпечення

У розробці програмного забезпечення вимоги, як правило, виявляються та конкретизуються перед їх впровадженням. Вимоги можуть бути вказані в різних видах артефактів і на різних рівнях формалізації (тобто необмежена природна мова, дисциплінована документація або формальна нотація) [44]. Застосування методів, спрямованих на підтвердження вимог, може залежати від рівня формалізації їхньої специфікації. Зокрема, концептуальні схеми, визначені в UML, є формальними специфікаціями функціональних вимог, і їх перевірка є основною метою підходу до тестування концептуальної схеми.

Перевірка концептуальних схем програмного забезпечення була темою, яка розглядалася в літературі. Роботу, пов'язану з цією дисертацією, можна проаналізувати в трьох вимірах: (1) область, тобто вид моделі, яка підлягає перевірці; (2) тип методу, який використовується для виконання підтвердження; і (3) цільова якість CS, покращена валідацією. Тепер ми розглядаємо репрезентативний набір існуючих методів перевірки концептуальних схем відповідно до вищезазначених параметрів.

У цьому розділі ми аналізуємо, як специфікації вимог можна перевірити в концептуальних схемах. Пов’язану роботу можна проаналізувати з трьох точок зору (рис. 2.1).

Домен. Відноситься до типу моделі, яка використовується для перевірки.

Мета якості: Посилається на ціль якості CS, яку необхідно покращити за допомогою перевірки.

Метод: стосується типу методу, який використовується для перевірки.

У решті цього розділу ми коротко опишемо ці розміри.

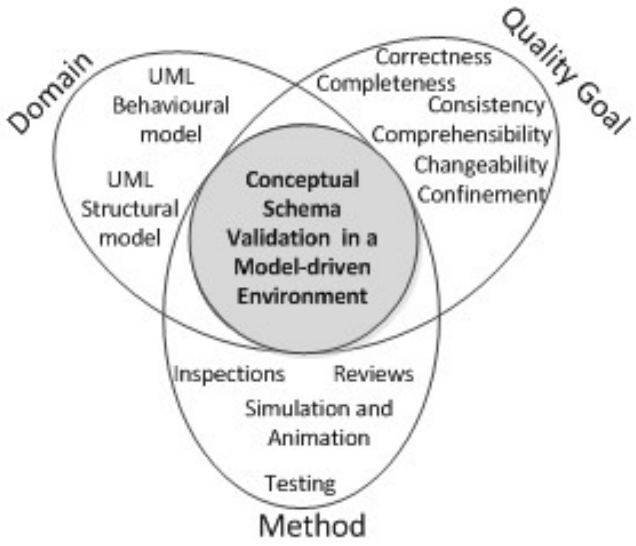

Рис. 2.1. Три простори для валідації концептуальної схеми

2.1.1. Домен

Розмір предметної області стосується типу моделі, що підлягає перевірці. У контексті програмного моделювання центром перевірки може бути структурна модель, поведінкова модель або обидві.

Що стосується першої групи, лише кілька робіт [45 - 47] аналізують структурні моделі окремо від моделей поведінки. Ці роботи пов'язані зі статичними методами, такими як огляд та перевірки [48].

У другій групі є дослідницькі пропозиції, присвячені проблемі валідації лише поведінкових моделей. Наприклад, у контексті UML існують роботи, які зосереджуються на перевірці лише діаграм діяльності [49], діаграм кінцевого автомата [50] і діаграми кінцевого автомата з діаграмою діяльності [40]. Решта робіт вимагають обох типів моделей (структурної та поведінкової) для підтвердження вимог до CS, наприклад, діаграми класів, включаючи операції та OCL; діаграма класів, діаграма взаємодії та діаграми діяльності; клас і послідовність тощо.

2.1.2. Мета якості

Цей вимір відноситься до цілі якості, яку потрібно покращити за допомогою перевірки. Кілька цілей якості, таких як послідовність, повнота,

зрозумілість і обмеженість, можна оцінити за допомогою ручної перевірки та, а також за допомогою контрольних списків [18].

До інспекцій повинні бути залучені як експерти з моделювання, так і нетехнічні експерти; особливо для оцінки аспектів зрозумілості та конфайнменту. Технології OORT (методи об'єктно-орієнтованого читання) є прикладом методів систематичної перевірки для перевірки («порівняння») діаграм UML одна з одною на повноту та узгодженість (вертикальну та горизонтальну) [50].

З іншого боку, роботи з тестування в основному спрямовані на повноту CS шляхом підтвердження вимог. Однак семантичну коректність, обмеженість і змінність можна покращити шляхом аналізу охоплених і не охоплених елементів (зайвих елементів) за допомогою тестових випадків.

Оскільки інформація з окремих діаграм (тобто структурної та поведінкової) повинна бути об'єднана з метою тестування, узгодженість між цими діаграмами має бути розглянута перед процесом тестування. Таким чином, якщо процес тестування підтримується інструментом для виконання CS, тоді некоректні дефекти виявляються синтаксичним аналізатором на попередньому етапі тестування, так що мета синтаксичної правильності також покращується.

2.1.3. Метод

Розмір методу відноситься до типу методу, який використовується для виконання перевірки. Для аналізу моделі можна використовувати різні методи. Їх можна класифікувати на статичні/динамічні та формальні/неформальні.

Далі ми розглядаємо методи перевірки вимог, які можуть бути застосовані до концептуальних схем. Відгуки та перевірки — це загальні методи, які можна застосовувати до інших типів специфікацій вимог. Інші були спеціально запропоновані для перевірки концептуальних схем, таких як тестування, моделювання та анімація. Крім того, ми коротко розглядаємо

репрезентативний набір підходів до верифікації, які можна використовувати в поєднанні з методами перевірки для забезпечення виконання процесу V&V.

Подібними методами в цьому відношенні є перевірки та перегляди специфікацій вимог [44]. Перевірки не потребують формальних специфікацій вимог. Однак, коли напівформальні або формальні специфікації є артефактами вимог, що перевіряються, процес може бути більш чітким, структурованим і простежуваним.

Концептуальні схеми визначають функціональні вимоги, і їх також можна перевірити та переглянути. Однак, оскільки про валідацію вимог важко судити, лише перевіряючи моделі, потрібна модель із виконуваними властивостями, щоб оцінити їх і виявити потенційні неправильні уявлення, виражені в моделі.

Методи, які виконують CS за допомогою моделювання та анімації надають користувачам засоби для виявлення невідповідностей і виконання операцій специфікації з параметрами, наданими користувачами, таким чином обчислюючи значення виходу параметри та новий стан системи. Ідея анімації концептуальних схем для цілей перевірки описують концептуальну мову (CPL) і інструмент, який генерує прототип зі схеми CPL, який можна протестувати. Згенерований прототип дає змогу побудувати стан Інформаційної бази, виконувати перевірки узгодженості та ставити запитання щодо вмісту Інформаційної бази.

Кілька інструментів підтримують редакції, моделювання та анімацію моделей UML. Наприклад, інструмент USE отримує діаграму класів UML і набір декларативних операцій і може перевіряти структуру/поведінку відповідно до очікувань модельєра/дизайнера (таких як узгодженість моделей UML і незалежність обмежень OCL) за допомогою анімації. У цьому контексті автори [42] представляють інструменти, що забезпечують графічну візуалізацію симуляцій на діаграмах активності, що підсвічують активні стани та можливі переходи, у поєднанні із засобами візуалізації та запису слідів виконання.

У контексті MDD методи тестування також застосовувалися для тестування моделей. У цих підходах досліджуваний артефакт є моделлю, а не вихідним кодом. Тестування є частиною процесу валідації та верифікації, де концептуальна схема працює в контрольованих умовах, щоб: (1) перевірити, що вона поводиться, як зазначено; (2) виявляти дефекти та (3) перевіряти вимоги користувача [12].

Оскільки функціональне тестування є методом перевірки, який розглядається в цій дипломній роботі, у наступному розділі ми підсумовуємо пов’язану роботу щодо генерації, вибору та пріоритезації тестових випадків, необхідних для процесу тестування.

Отже, валідація концептуальної схеми на основі тестування — це область досліджень, яка допускає нові методи та техніки, стикається з такими проблемами, як створення тестових випадків з використанням зовнішньої інформації щодо концептуальних схем (тобто вимог), вимірювання можливої автоматизації, вибір і пріоритезація тестових випадків і потреба в прибутковому допоміжному інструменті, що використовує стандартну семантику, своєчасний зворотний зв’язок для підтримки процесу забезпечення якості програмного забезпечення та полегшення прийняття рішень на основі аналізу та інтерпретації результатів.

2.2. Методологія перевірки на основі тестування для концептуальних схем у середовищі, керованому моделлю

Як згадувалося, метою роботи є надання набору методів, які допоможуть розробникам моделей (аналітикам/дизайнерам) і тестувальникам покращити якість концептуальних схем.

У цьому розділі описується методологія перевірки на основі тестування для концептуальних схем у середовищі, керованому моделлю, яка долає виклики: перевірка вимог на ранній стадії (тобто концептуальні схеми) і автоматичне створення тестових випадків для концептуальних схем. Щоб

подолати ці виклики, дана методологічна структура підтримує використання методів моделювання, керованих моделями (MDE), які передбачають інтенсивне використання моделей для підтримки різних етапів запропонованої структури. Ми вважаємо, що використання MDE зменшує складність запропонованого підходу перевірки, оскільки дозволяє розробникам моделей і тестувальникам працювати на високому рівні абстракції, а також підвищує автоматизацію та повторне використання. У нашому підході рівень абстракції підвищується, дозволяючи розробникам моделей визначати вимоги за допомогою методу розробки вимог, який забезпечує концептуальні конструкції високого рівня.

Автоматизація підвищується за допомогою трансформацій моделей, які приймають моделі вимог як вхідні дані та автоматично генерують реалізації тестових випадків, які інтегруються в середовище тестування, яке виконує їх на виконуваних концептуальних схемах, щоб допомогти тестувальникам під час перевірки вимог концептуальних схем. Повторне використання збільшується завдяки дозволу тестувальникам повторно використовувати частини тестових моделей для створення тестових випадків для інших типів концептуальних схем (тобто концептуальних схем на основі ОО-методу). Таким чином, ми забезпечуємо швидке створення тестових моделей, а також автоматизуємо реалізацію тестових випадків за допомогою композиції багаторазово використовуваних компонентів тестової моделі.

Відповідно до [32] ми розуміємо структуру як цілісний і стислий опис концепцій і методів, що стосуються певної області. У цій роботі, як згадувалося в розділі 1, ми пропонуємо структуру, яка допоможе аналітикам/дизайнерам покращити якість їхніх концептуальних схем. Наша система перевірки на основі моделі забезпечує середовище виконання для автоматизації тестових сценаріїв (тобто тестових сценаріїв) і, таким чином, надає користувачеві різноманітні переваги, які допомагають проектувати, генерувати, вибирати, виконувати автоматизовані тестові сценарії та звітувати про них.

2.2.1. Етапи методології

На рисунку 2.2 графічно зображено структуру валідації, засновану на тестуванні на основі моделі, запропоновану в цій роботі, яка описана в наступних розділах.

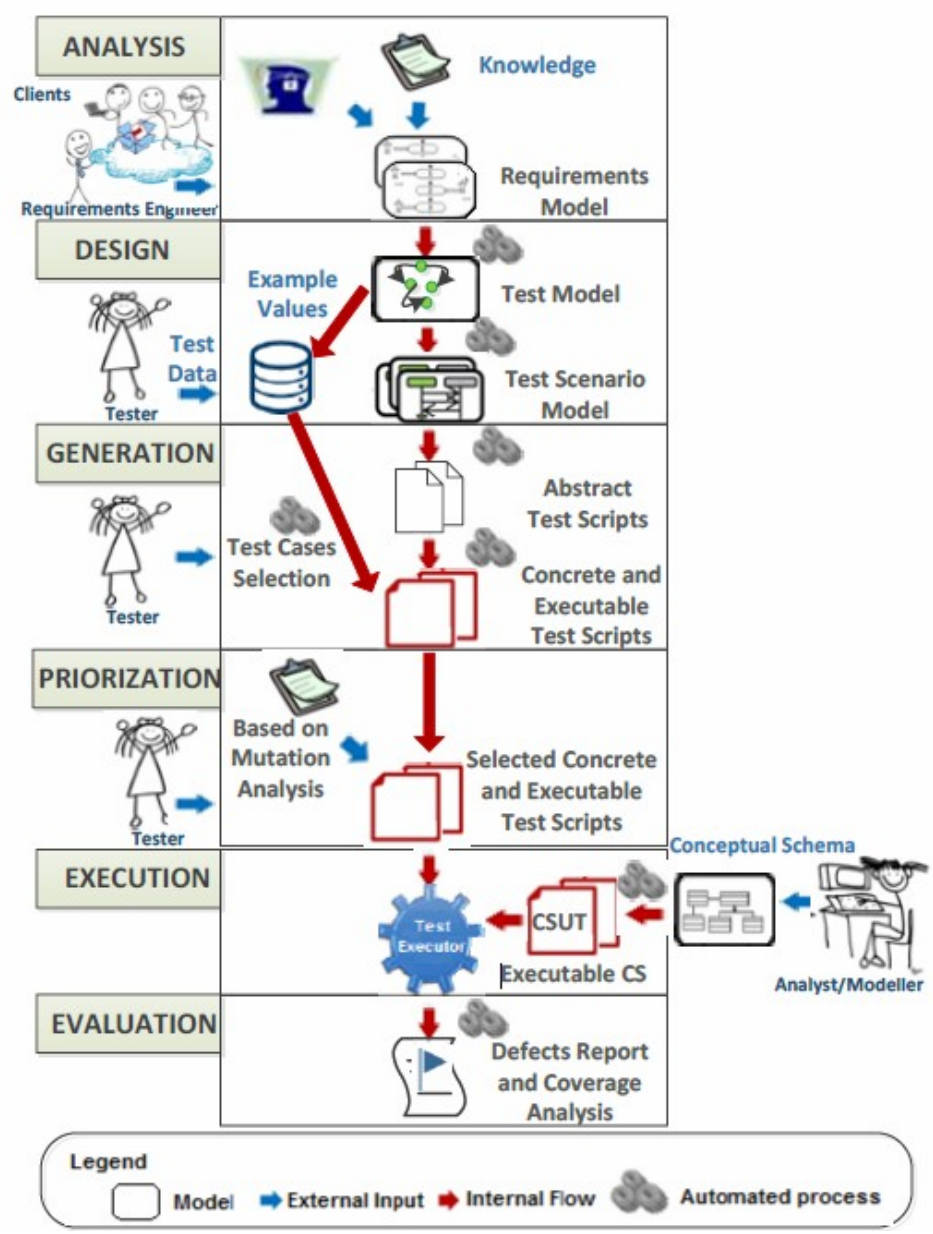

Рис. 2.2. Огляд етапів методології перевірки

2.3. Представлення аналітичної фази валідації схеми

У цьому розділі ми зосередимося на аналітичній фазі нашого фреймворку валідації. Ця фаза вимагає вивчення вимог та спілкування з

різними стейкхолдерами для отримання комунікативних взаємодій між інформаційними системами та їх середовищем.

На цій фазі розуміються та моделюються програмні вимоги у моделі вимог за допомогою аналізу комунікації [21]. Модель вимог є екземпляром Метамоделі Вимог, яка описує системні вимоги на рівні бізнесу і специфікується бізнес-експертами та системними аналітиками.

2.3.1. Специфікація вимог на основі комунікаційного аналізу

Наша перша причина для використання комунікаційного аналізу полягає в тому, щоб отримати єдину модель для визначення функціональних можливостей інформаційної системи (IS) і створити відповідні тестові випадки. Таким чином уникають використання різних артефактів аналітиками вимог, тестувальниками та розробниками, що полегшує їх роботу. Оскільки послідовність подій описує очікуваний обмін повідомленнями між актором і системою, це можна використовувати для визначення тестових випадків. Зокрема, у той час як комунікативні події вказують на дії, які мають бути виконані повністю та безперервно за певних обмежень, структури повідомлень для кожної комунікативної події містять посилання на задіяні типи, які представляють акторів або бізнес-концепції, зв’язки між ними та параметризовані повідомлення з типами даних, існуючими в концептуальній схемі системи (діаграма класів і кінцеві автомати). Однак це змушує аналітика вимог бути точним і суворим у семантиці, наданій кожній концепції, і, отже, може бути не так легко створити. Щоб зменшити цю складність, ми використовуємо існуючий інструмент редактора [15], який є доменно-специфічною мовою, щоб створити діаграму комунікативної події (CED) і представити структуру повідомлення для кожної комунікативної події.

Наша друга причина полягає в тому, що тестування на основі вимог [3], зокрема тестування на основі моделі [16], все частіше використовується.

Таким чином, існує потреба в систематичному підході до створення тестових випадків на основі моделі вимог.

Наша третя причина полягає в контексті MDD, де можна отримати тестову модель із моделі вимог за допомогою перетворень моделі, щоб процес міг продовжувати генерувати тестові випадки виконуваних файлів. Це означає, що коли в модель вимог вноситься модифікація, автоматично повторно генерується не тільки тестова модель, але й конкретні тестові випадки.

Нарешті, CA було інтегровано в UML-сумісну структуру розробки, керовану моделлю [6], а також стратегію трансформації моделі, щоб отримати початкові версії концептуальних схем із моделей вимог комунікаційного аналізу. Це означає, що він включає в себе примітиви (примітиви шаблону специфікації події), які потрібні керованому моделлю методу (достатньо дрібні, щоб бути представленими безпосередньо в коді), щоб виразити структуру та динаміку IS.

2.3.1. Моделювання вимог на основі комунікаційного аналізу

Комунікаційний аналіз пропонує кілька методів моделювання для моделювання бізнес-процесів і специфікації вимог. Діаграма комунікаційних подій (CED - Communicative Event Diagram) описує бізнес-процеси з комунікаційної точки зору. На рисунку 2.3 показано два CED моделі CA для випадку Video Club (тобто управління користувачами та прокатом фільмів відповідно). CED складається зі структурованої послідовності, визначеної відношеннями пріоритету між комунікативними подіями (CE) (заокруглені прямокутники на рисунку 2.3). CE — це дія, пов’язана з інформацією (отримання, зберігання, обробка, пошук та / або розповсюдження). CE виконується повністю та безперервно, коли є зовнішній стимул для системи (тобто вхід користувача в систему). CE може бути спеціалізований за допомогою варіантів подій, які є альтернативними подіями, які визначають

шляхи в CED (наприклад, на рисунку 2.3 розв'язка входу спеціалізується на вхід прийнято або вхід відхилено).

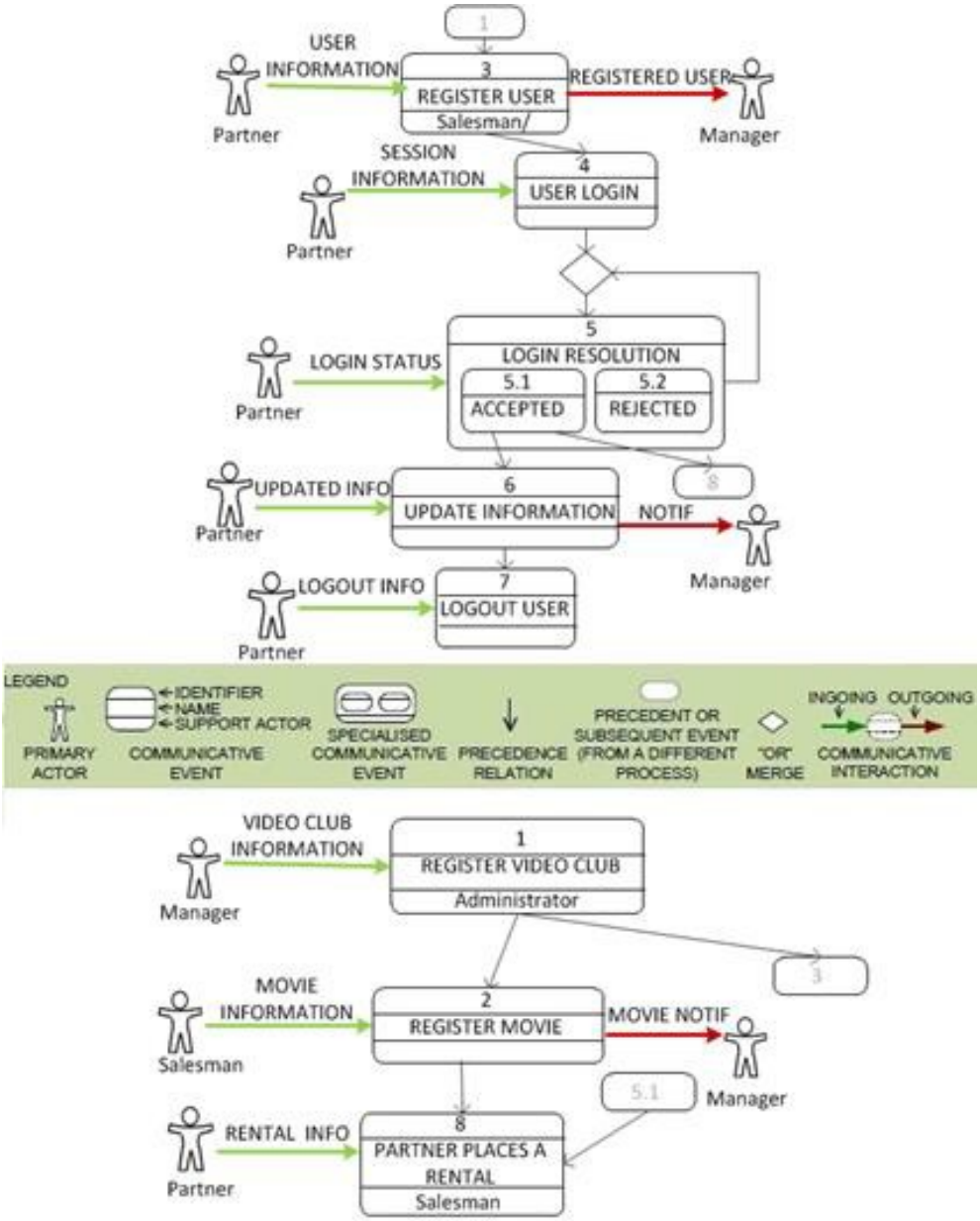

Рис. 2.3. Витяг моделі CA для корпусу Video Club.

CED має зв'язки, щоб визначити вхідні та вихідні комунікативні взаємодії та три ролі актора:

1) основні ролі (тобто основний актор), які запускають CE та надають вхідну інформацію,

2) ролі одержувача (тобто актор-одержувач), які повинні бути поінформований про настання події;

3) ролі інтерфейсу (тобто актор підтримки), який відповідає за редагування та введення вхідної інформації. У цьому прикладі в реєстраційному користувачі CE партнер виконує основну роль, менеджер виконує роль отримувача, а продавець виконує роль інтерфейсу. Щоб детально описати автор [21] запропонував використовувати шаблони специфікації подій.

Шаблон специфікації події структурує вимоги [21] і є технікою текстової специфікації, яка використовується для опису як вхідних, так і вихідних повідомлень, що передаються до IS у комунікативній події. Він використовує структуру повідомлення для визначення інформації, яка передається під час події.

Шаблон складається із заголовка та трьох категорій вимог: вимоги до контакту, комунікаційного вмісту та вимоги до реакції. Заголовок містить інформацію про CE, таку як ідентифікатор події, ім’я, ціль , опис розповіді тощо. Контакт (вимоги, пов’язані з ініціюванням події суб’єктом, щоб повідомити щось до інформаційної системи, наприклад, попередні умови), повідомлення (вкажіть вміст повідомлення, яке передається до або з IS, наприклад, поля повідомлення, домен полів повідомлення, обмеження повідомлень); і вимоги до реакції описують, як ІС реагує на виникнення комунікативної події (наприклад, зберігає нові знання, робить нові знання та висновки доступними для відповідних акторів). Таким чином, ця категорія вимог включає бізнес-об’єкти, які реєструються (тобто лікування), і вихідні комунікативні взаємодії, які генеруються подією (наприклад, пов’язана поведінка та пов’язане спілкування), серед інших вимог. Наше дослідження охоплює тестування вимог до комунікативних подій. Далі в таблиці 2.1 показано спрощений шаблон специфікації події.

Структура повідомлення визначає інформацію, яка передається до або з інформаційної системи [105]. Таблиця 2.1 показує структуру повідомлення

для комунікативної події (тобто продавець реєструє фільм) у нашому прикладі. Наступні граматичні конструкції представляють інтерес для цілей виведення тестової моделі.

Таблиця 2.1.

Приклад шаблону специфікації події

7. REGISTER MOVIE INFORMATION

Goals: From the point of view of the information system, the objective of this event is to record the relevant information about the movie rents.
Description: When a partner rent movies, both the rent date and return date should be registered in the SI. More than one movie may be included in a rental. The rental price is calculated as a derivative, adding the movie prices that make up the rental.

Contact requirements

Primary actor: Partner
Communicational channel: In person
Temporal restrictions: none
Frequency: none

Communicational content requirements

Support actor: Salesman
Communication Structure: (see the next partial view of a Message Structure)

FIELD	OP	DOMAIN	EXAMPLE VALUE
RENTAL =			
< id rental +	g	Number	7260
pick up date +	i	date	18-05-2016
return date +	i	date	20-05-2016
total +	i	money	2.5
Partner +	i	PARTNER	User100, Jorge Vidal
RENTALLINES =			100, Valencia,...
{RENTALLINE =			
<rental number +	i	Number	250
price +	d	Money	this.movie.rental_price
Movie >	i	MOVIE	100, Everest,...
MOVIESTATUS =			
<status +	i	Text	Rented
Movie >>}	i	MOVIE	100, Everest,...

Structural restrictions: One rent can have many movies.
Contextual restrictions: Rental is identified by the id rental.

Reaction requirements

Treatments: The rent lines are recorded and they are assigned to the movie rent. Movie status is updated to “Rented”.
Linked behaviour: The rent is related to a partner.
Linked communications: none

Підструктура — це елемент, який є частиною структури повідомлення. Наприклад, Partner, RENTALLINES, RENTALINE, Movie, MOVIESTATUS є підструктурами RENTAL. Початкова підструктура - це перший рівень

структури повідомлення. У нашому випадку RENTAL = <ідентифікатор оренди + дата отримання + дата повернення + загальна сума + партнер + RENTALLINES, MOVIESTATUS>.

2.4. Дизайн тесту та процес генерації тестів

На цьому етапі інформація про базу тестування береться з моделі вимог і перетворюється на тестову модель (ТМ) з тестовими умовами/елементами (щось, що можна перевірити, наприклад, служби, тригери, твердження та посилання), упорядкованими за відношеннями пріоритету, який генерує впорядкований граф. Ця модель відповідає метамоделі тестової моделі (ТММ).

Потім у тестовій моделі визначаються різні шляхи для створення моделі тестових сценаріїв із тестовими елементами, об’єднаними в абстрактні тестові випадки. Тестові випадки є абстрактними в тому сенсі, що вони не містять конкретних об’єктів.

2.4.1. Тестові дані

Для специфікації тестових значень дані витягуються з тестової моделі та зберігаються в базі даних. Ці значення є прикладами значень, переданих до тестової моделі з моделі вимог. Іншим джерелом даних є значення, які безпосередньо вводяться в базу даних користувачем (розробником моделі або тестувальником). Нарешті, веб-стратегія створення дійсних тестових рядків з використанням регулярних виразів, наданих користувачем (розробником моделі або тестувальником), може бути використана для створення тестових значень.

2.4.2. Генерація тестів

Тестові випадки також можуть бути згенеровані шляхом переходу від батьківського кореня до дочірнього вузла за допомогою класичного

алгоритму пошуку шляхів або обходу графа. Коли пройдено всі вузли на шляху від батьківського до дочірнього вузла, це вважається одним тестовим сценарієм. Усі вузли повинні бути охоплені, щоб переконатися, що всі потоки в додатку охоплені. Один потік вважається одним тестовим сценарієм. Набір тестів для CS — це набір з одного або кількох сценаріїв тестування. Кожен тестовий сценарій - це історія, яка складається з ще одного тестового випадку. На цьому етапі абстрактні тестові сценарії генеруються з моделі тестового сценарію. Потім конкретні та виконувані тестові випадки (сценарії) генеруються з моделі тестового сценарію, щоб описати, що система має робити з вхідними даними, взятими з бази даних, а також оракул і цілі тестового випадку. Усе це робиться за допомогою перетворень «модель-модель» і «модель-текст» після розроблення, керованого моделлю. Деталі генерації, керованої моделлю, обговорюються в розділі 6. У наступних розділах ми підсумовуємо дизайнерські рішення, які розглядаються для генерації тестових випадків.

2.4.3. Вибір тестового випадку

Оскільки процес генерації генерує багато тестів для покриття різних сценаріїв тестування, нам потрібно знати, які тести слід інвентаризувати, а які слід видалити. Оскільки тестові сценарії можуть використовувати деякі спільні оператори (загальний шлях у моделі тестового сценарію), процес генерації тестових випадків може отримати велику кількість дублікатів тестових випадків. Критерієм виявлення дублікатів тестових випадків є відповідність оператору Test. Потім ми пропускаємо генерацію дубльованих тестів і зберігаємо як тип згенерованого тесту, так і дублікати тестів для звіту в результаті процесу генерації.

2.4.4. Досягнуті цілі якості

Тестові випадки в основному спрямовані на перевірку двох цілей якості CS [9]: правильність (охоплює як синтаксичну правильність - правильний

синтаксис або правильне оформлення, так і семантичну правильність - правильне значення та зв'язки щодо знань про предметну область) і повноту (тобто вся необхідна інформація визначена в CS). Однак також розглядаються інші цілі якості, такі як послідовність, обмеженість, зрозумілість і змінність.

Типи тестів визначають загальні типи очікувань, які необхідно вказати в тестових випадках для тестування концептуальних схем. У концептуальному моделюванні (фрагмент) часу життя інформаційної системи є послідовність станів CS, яка представляє моментальний знімок стану домену як екземпляра концептуальної схеми. У нашому підході ми розглядаємо тестові випадки для тестування концептуальних схем як послідовність станів CS (тобто конкретної історії користувача) разом із формалізованими очікуваннями (тобто тестові оракули та цілі тестування) щодо цих станів. Очікується, що ця послідовність станів буде успішно виконана, якщо необхідні знання правильно та повністю визначені в концептуальній схемі.

Потім тестувальник може вибрати тип тесту:

- Partial (тільки позитивні тестові випадки): цей тип тестового прикладу використовує твердження для перевірки виникнення події та вмісту об’єктів CS.
- Complete, який додає тестові випадки (отже, позитивні тестові випадки) з деякими негативними умовами, такими як значення поза діапазоном на основі розділів змінних, які можуть бути отримані з інформації CS, порушення обмежень, порушення мінімальної кардинальності та порушення унікального значення для змінних класу. Таким чином ми перевіряємо відсутність недійсної події.

2.4.5. Критерії генерації тестів

Крім того, для вибору тестових випадків, які мають бути згенеровані, наша структура застосовує набір критеріїв генерації, на основі елементів

покриття (тобто класів, асоціацій та узагальнень) у структурній частині, а також поведінкових частина (умова, усі шляхи повідомлення) (табл. 2.2).

Таблиця 2.2.

Критерії генерації тестів для концептуальної схеми на основі UML

Association-end multiplicity (AEM) criterion Given a test suite T and a test model TM, T must cause each representative multiplicity-pair in TM to be created.
Generalization (GN) criterion Given a test suite T and a test model TM, T must cause every specialization defined in a generalization relationship to be created.
Class attribute (CA) criterion Given a test suite T, a test model TM, and a class C, T must cause a set of representative attribute value combinations in each instance of class C to be created.
Condition coverage (Cond) criterion Given a test suite T and test model TM, T must cause each condition in each decision in TM to evaluate to both TRUE and FALSE.
Full predicate coverage (FP) criterion Given a test suite T and test model TM, T must cause each clause in every condition in TM to take the values of TRUE and FALSE while all other clauses in the predicate (condition) have values such that the value of the predicate will always be the same as the clause being tested.
Each message on link (EML) criterion Given a test suite T and diagram Class (DC), T must cause each message on a link connecting two objects in CD to be executed at least once.
All message paths (AMP) criterion Given a test suite T and test model TM, T must cause each possible message path (sequence of message numbers) in TM to be taken at least once.
Collection coverage (Coll) criterion Given a test suite T and test model TM, T must test each interaction with collection objects of various representative sizes at least once.

2.4.6. Виведення цілей тестування

У нашому фреймворку критерії генерації тестів та типи тестів можуть використовуватися для виведення цілей тестування. На рисунку 2.4 показано деякі приклади цілей тестування для Video Club CS на основі критерію Coll.

Наприклад:

(i) Критерій Coll може бути пов’язаний з цілями тестування для позитивних тестових випадків, які вимагають, щоб система була приведена

до певної конфігурації, що має певну кількість об’єктів у колекції, що з’являється в тестовій моделі;

(ii) Критерій EML може бути використаний для генерації цілі тестування для позитивного тестового випадку, який передбачає конкретні зв’язки, які мають бути використані під час тестів;

(iii) Критерії CA та Cond можуть бути використані для виведення значення атрибута в кожному екземплярі класу rental, який потрібно створити;

(iv) Критерії FP та Cond можуть бути використані для виведення цілей тестування для негативних тестових випадків, які передбачають значення для певної умови;

(v) Критерій CA може бути використаний для генерації цілі тестування для негативного тестового випадку, який перевіряє значення атрибута в кожному екземплярі класу videoclub, який потрібно створити;

(vi) Критерій AMP може бути використаний для визначення цілі тестування для позитивного тестового випадку, який передбачає конкретні шляхи, які мають бути використані під час тестів.

i. Validate the Object 'videoclub_' was created (test case positive)
ii. Validate the link 'videoclub_movie.createLink(videoclub_,movie_);' with a valid value (test case positive)
iii. Validate the derived Attribute 'context rental inv property_total_derivation:' (test case positive)
iv. Validate unique value: 'context VideoClub:: new_videoclub() post: VideoClub->isUnique e (e.id_videoclub)' (test case negative)
v. Validate a value above the upper limit 'context videoclub:: new_VideoClub() pre: p_atrid_videoclub<=10000' (test case negative)
vi. Validate the 'line 28' with valid values (test case positive)

Рис. 2.4. Приклад тестових цілей

2.5. Конкретні та виконувані тестові випадки

У нашому тестовому фреймворку ми адаптуємо термінологію UTP і вважаємо, що тестовий випадок є специфікацією одного випадку для тестування концептуальної схеми, включаючи те, що тестується, який вхід, результат і за яких умов. Потім тестові випадки, згенеровані нашою пропозицією, демонструють такі властивості:

- Тестовий випадок складається з фікстури та одного або кількох тверджень, які виконують один із тестів, застосовних до концептуальних схем, таких як тестування тверджень про появу або непояву події. Фікстура - це набір тверджень (наприклад, створення об’єкта або зв’язку, виконання методу об’єкта), які створюють стан CS і визначають значення змінних концептуальної схеми.

- Оракул і мета тестування кожного тестового випадку виводяться з типу тестових випадків, вибраних на попередньому етапі. Очікуване значення (оракул) для позитивних тестових випадків є assertionEqual або assertionTrue рівним "true", а з негативними умовами assertionFalse має бути істинним, інакше тестовий випадок не пройшов.

- Кожне виконання тестового випадку починається з виконання фікстури.

- Припускається, що виконання кожного тестового випадку починається з порожнього стану. З цим припущенням тестові випадки CS є незалежними один від одного, і порядок їх виконання тому не має значення.

В ALF виконуваний тестовий випадок є діяльністю, яка забезпечує специфікацію параметризованої поведінки як скоординовану послідовність підпорядкованих одиниць ALF. Це фундаментальний механізм для моделювання поведінки в ALF.

Кожен конкретний тестовий випадок має ім’я і складається з набору тверджень (див. рис. 2.5).

```
private import namespace::*;
public import Library;
//Goal: … <oracle< ... (<test type>)
activity TestCaseName () {
…
assert …
}
```

Рис. 2.5. Структура тестового випадку

Останнє твердження конкретного тестового випадку є твердженням. У цьому розділі ми описуємо синтаксис і семантику п’яти типів тверджень, пов’язаних із тестуванням концептуальних схем:

- Твердження, які оновлюють інформацію про об’єкти CS,
- Твердження, які стверджують про появу подій,
- Твердження, які стверджують про непояву подій,
- Твердження, які стверджують про вміст об’єктів CS.

2.5.1. Оновлення інформації про об’єкт концептуальної схеми

Коли починається виконання тестового випадку, припускається, що інформація CS є порожньою, і тому нам потрібно поступово налаштувати різні стани CS, щоб перевірити стан, який не може бути досягнутий дійсними подіями. ALF включає твердження, які можна використовувати для явного налаштування стану CS у тестовому випадку. Ми описуємо їх нижче, використовуючи приклади на основі фрагмента схеми Video Club (див. рисунок 2.6).

Ми визначаємо, що entityID є новим екземпляром будь-якого типу сутності за допомогою наступного твердження:

```
EntityType entityID = new EntityType();
```

Щоб визначити, що значення атрибута att сутності entityID дорівнює val (де val є коректним виразом OCL), ми пишемо:

```
entityID.att = val;
```

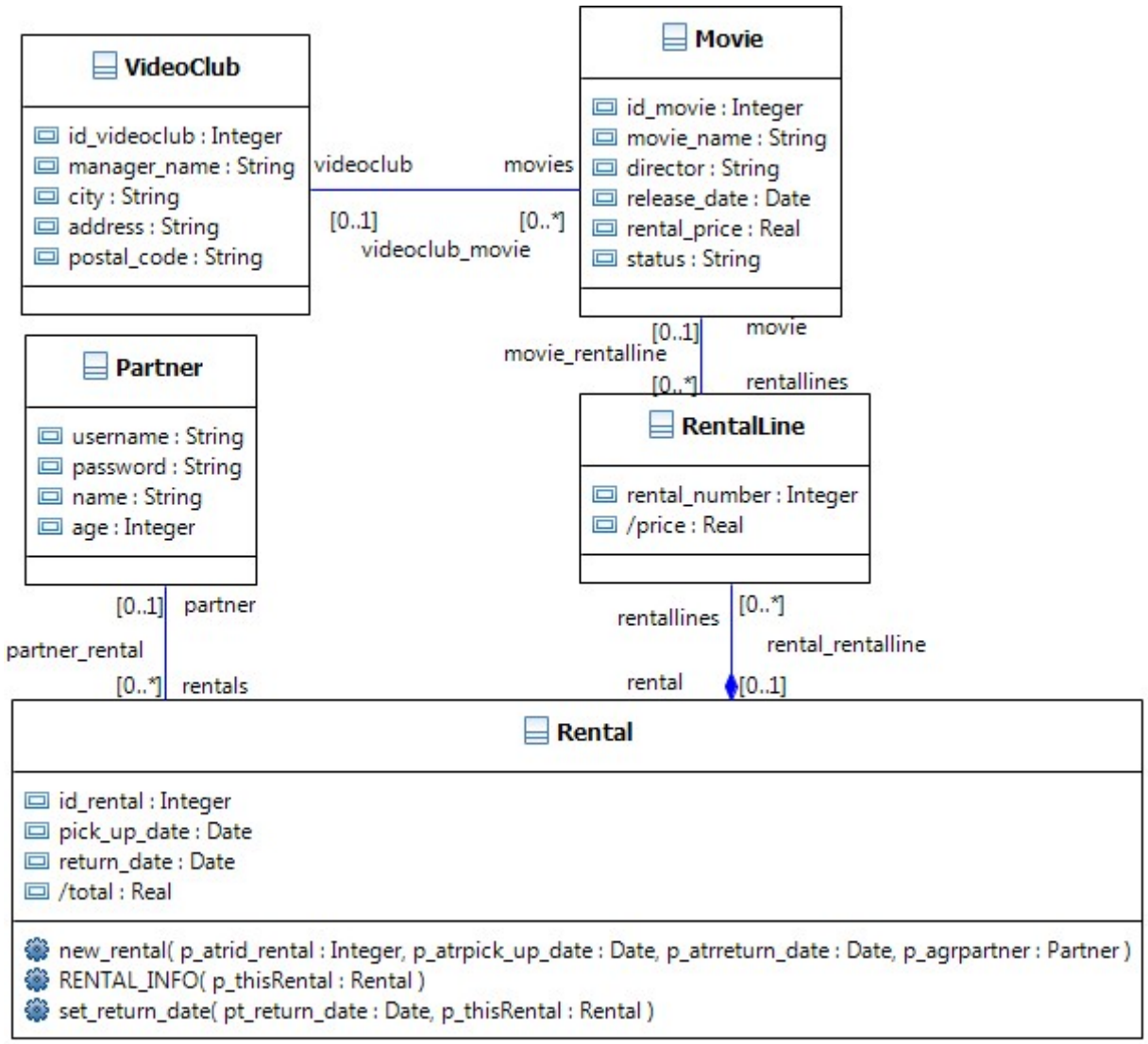

Рис. 2.6. Діаграма класів UML для концептуальної схеми Video Club

Типи att і val повинні бути сумісними; інакше вердикт тестового випадку, в якому з'являється твердження, є невизначеним.

Часто зручно в одному твердженні вказати створення нової сутності entityID як екземпляра типу сутності EntityType. Синтаксис наступний:

```
entityID = new EntityType(parameter1=value1, ...,
                          parameterN=valueN);
```

Використовуйте цей код обачно.

де entityID має бути новим ідентифікатором, а valuei є значеннями або виразами. Наприклад, для створення екземпляра відеоклубу:

```
videoclub_ = new VideoClub(p_atrid_videoclub=100,
p_atrmanager_name="Jose Vicente Vidal", p_atrcity="Valencia",
p_atraddress="Guardia Civil 21", p_atrpostal_code="46020");
```

Екземпляри n-арної асоціації UML Assoc з ролями r1, ..., rn створюються за допомогою наступного твердження:

```
AssociationName.createLink(entityA, entityB);
```

Де entityA та entityB повинні бути кінцевими членами асоціації. Наприклад, щоб створити екземпляр асоціації videoclub_movie:

```
videoclub_movie.createLink(videoclub_, movie_);
```

Сутності можна видалити за допомогою наступного твердження:

```
objectID_.destroy();
```

Наприклад, щоб видалити екземпляр відеоклубу.

```
videoclub_.destroy();
```

Видалення сутності означає видалення її атрибутів і зв’язків, у яких вона бере участь. Однак, зауважте, що анотація композиції знаходиться на кінці частини складеної асоціації. Наприклад:

```
assoc rental_rentaline {
    public 'rental': Rental[1];
    public 'rentallines': compose RentalLine;
```

```
}
```

Тобто, у вищевказаній асоціації Rental є складовою, а Rentalline є частиною. Таким чином, коли екземпляр класу Rental знищується, якщо існує зв'язок асоціації rental_rentalline з цим об'єктом на одному кінці, то цей зв'язок і екземпляр Rentalline на іншому кінці також будуть знищені.

2.6. Пріоритезація тесту

Оскільки процес тестування керує багатьма тестами, нам потрібно знати, наскільки якісним є тест. Щоб виконати цю роботу ефективно, нам потрібно знати пріоритезацію тестових випадків, які тестові приклади потрібно виконати? Які є критичними? Однією з проблем у розробці тестів для оцінки якості тестових випадків є те, що реальні артефакти програмного забезпечення відповідного розміру, включаючи реальні помилки, важко знайти та підготувати відповідним чином (наприклад , шляхом підготовки правильних і помилкових версій). Навіть коли артефакти програмного забезпечення з реальними помилками доступні, ці помилки зазвичай не є достатньо численними, щоб дозволити експериментальним результатам досягти статистичної значущості.

У цьому контексті тестування на мутації є одним із способів оцінки якості набору тестів, щоб визначити пріоритетність зусиль у критичних. Цей метод вводить штучні дефекти або зміни в CS (покоління мутантів) і перевіряє, чи набір тестів «достатньо хороший» для виявлення цих штучних дефектів. Штучні помилки можуть бути створені автоматично за допомогою набору операторів мутації (MO) для зміни (тобто мутації) деяких частин артефакту програмного забезпечення. Мутанти можна класифікувати на два типи: мутанти першого порядку (FOM) і мутанти вищого порядку (HOM). FOM генеруються шляхом застосування операторів мутації лише один раз.

HOMs генеруються шляхом застосування операторів мутації більше одного разу [13].

Якщо припустити, що артефакт програмного забезпечення, який мутується, є синтаксично правильним, оператор мутації повинен створити мутант, який також є синтаксично правильним. Кожна несправна версія артефакту або мутант виконується проти набору тестів. Співвідношення виявлених мутантів до загальної кількості нееквівалентних мутантів відоме як «оцінка мутацій» і вказує на те, наскільки ефективними є тести з точки зору виявлення помилок. Таким чином, критерії адекватності мутаційного тесту можуть допомогти оптимізувати процес тестування [14]. Його можна використовувати для визначення набору тестів - вибору тестів із величезного пулу тестів. Умовою вибору тесту є виявлення дефектів у мутованих артефактах програмного забезпечення.

У тестуванні на мутації найбільш критичною діяльністю є адекватне проектування операторів мутації, щоб вони відображали типові дефекти тестованого артефакту. Тому ми повинні розробити набір операторів мутації для концептуальних схем (CS) на основі діаграм класів (CD) уніфікованої мови моделювання (UML). Основна потенційна перевага операторів мутації полягає в тому, щоб точно описати мутанти, які можуть генерувати і, таким чином, підтримувати чітко визначений процес введення помилок.

Рисунок 2.7 ілюструє процес визначення операторів мутації. Як вхідні дані надано метамодель діаграми класів UML [40], типи дефектів у моделі на основі UML [43].

Кожен елемент діаграми класів UML було проаналізовано на основі типів дефектів, які можна впровадити. Потім усі оператори мутації були використані для генерації мутантів. Для того, щоб відкинути еквівалентні та недійсні мутанти, було виконано статичний аналіз і синтаксичний аналіз (за допомогою аналізатора ALF) мутантів. Потім було виконано процес відбору, щоб отримати список операторів мутації (тобто FOM і HOM) для використання мутації.

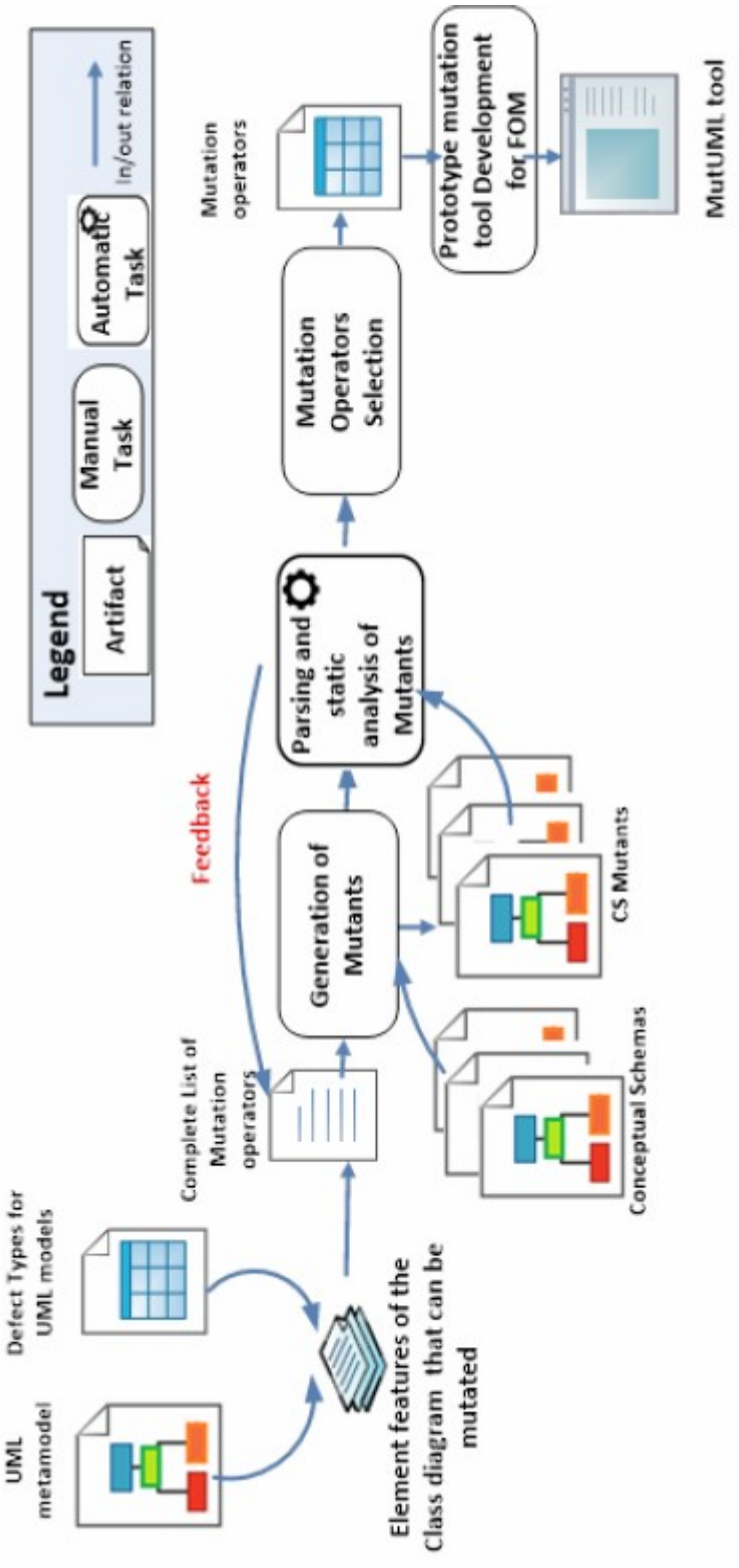

Рис. 2.7. Процес відбору операторів мутації

Нарешті, дослідники зібралися для прийняття рішень з двома основними цілями: 1) зосередитися на оцінці корисності операторів мутації для FOM і 2) автоматизувати генерацію мутантів та оцінити її можливість. У

роботі [43] представили класифікацію дефектів на рівні моделі, а в [45] описали процес вибору 18 операторів мутації зі списку з 50 для генерації мутантів першого порядку для CS на основі UML. На відміну від мутації на основі коду, наші оператори мутації базуються на характеристиках елементів CS на основі UML CD і, хоча деякі з запропонованих операторів виконують синтаксичні зміни на рівні обмежень, вони в основному зосереджені (тобто 41 з 50 операторів) на семантичних змінах високорівневих конструкцій CD. Наші оператори мутації класифікуються за елементом, на який впливає оператор, типом ін'єктованого дефекту та дією, необхідною оператору мутації для генерації дійсних мутантів (синтаксично коректних). Оскільки наша мета полягає у виборі операторів мутації для оцінки підходів до тестування, процес вибору операторів мутації був розділений на два ітерації.

Тому тестові випадки, які перевіряють кратність і обмеження, мають пріоритет у тестовому наборі, а також тестові випадки, які охоплюють повні тестові сценарії. Однак типи агрегації вимагають статичного аналізу для їх перевірки.

Висновки до розділу

Другий розділ детально аналізує методи та моделі валідації концептуальних схем у контексті процесів тестування. Він охоплює аспекти перевірки якості схем, методологію тестування та розробку тестових випадків, спрямованих на забезпечення їх надійності та відповідності вимогам.

Валідація концептуальних схем починається з розуміння домену, визначення мети якості та вибору відповідних методів перевірки. У розділі висвітлено, як правильне окреслення контексту (домену) та чітке визначення цілей якості забезпечують ефективність перевірки. Методологія тестування розглядається як поетапний процес, що включає підготовку до валідації, тестування та аналіз результатів. Особлива увага приділяється розробці

підходів, які інтегрують перевірку вимог із загальними етапами розробки програмного забезпечення, створюючи цілісну модель перевірки.

Розробка концептуальних схем передбачає специфікацію та моделювання вимог, де особливо важливим є комунікаційний аналіз. Він допомагає деталізувати взаємодію між учасниками процесу та визначити ключові елементи, які потребують перевірки. Важливим аспектом є створення тестів, які можна виконати безпосередньо на концептуальних схемах. Це включає оновлення інформації про об'єкти схем, перевірку зв’язків між компонентами та виявлення недоліків у логіці або структурі.

У підсумку, методи та моделі тестування концептуальних схем є важливим інструментом для забезпечення їхньої відповідності специфікаціям і запобігання помилкам у процесі розробки програмного забезпечення. Комплексний підхід до валідації дозволяє покращити якість схем, зменшити ризики та оптимізувати процес створення програмних рішень.

РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ МОДЕЛЕЙ ТА ЗАСОБІВ КОНЦЕПТУАЛЬНОЇ СХЕМИ ВАЛІДАЦІЇ НА ОСНОВІ ТЕСТУВАННЯ В МОДЕЛЬНО-ОРІЄНТОВАНОМУ СЕРЕДОВИЩІ

3.1. Виконувана концептуальна схема на основі діаграми класів UML

Оскільки тестові сценарії (інструкції тестового сценарію) мають виконуватися щодо концептуальної схеми, що тестується, нам потрібен виконуваний CS як вхідні дані для середовища тестування.

Діаграма класів (рис. 3.1) є основним блоком UML, який показує елементи системи на абстрактному рівні (наприклад, клас , клас асоціації), їхні властивості (ownedAttribute), відносини (наприклад, асоціація та узагальнення) та операції. В UML операція визначається шляхом визначення обмежень. На рисунку 3.1 показано фрагмент структури UML [40] для діаграми класів і виділено вісім елементів, що представляють інтерес для цієї роботи.

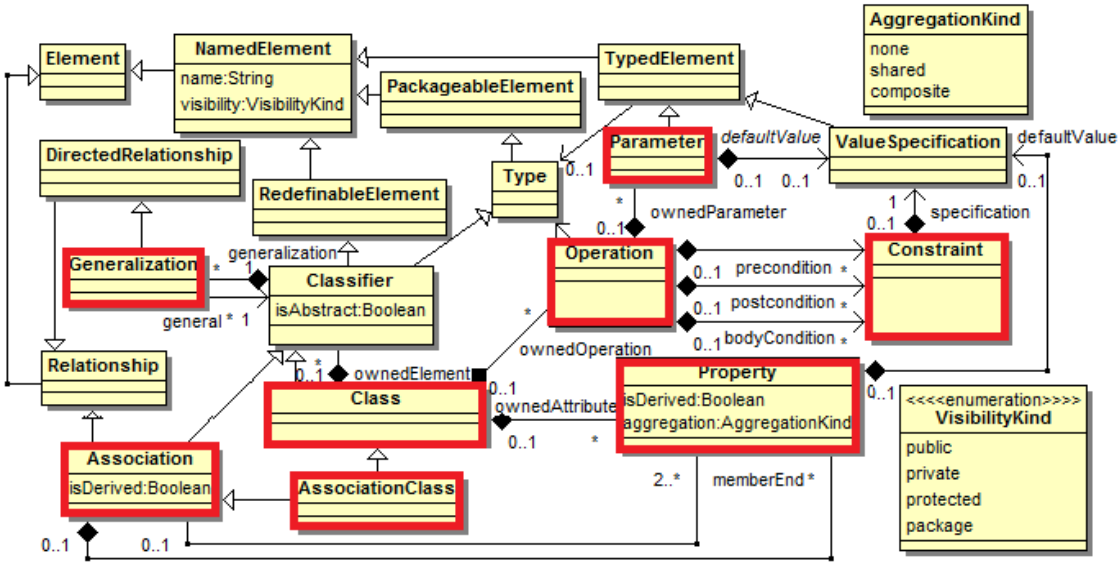

Рис. 3.1. Фрагмент метамоделі діаграми класів UML

Виконувана модель знаходиться на наступному вищому рівні абстракції, абстрагуючись як від конкретних мов програмування, так і від

рішень щодо організації програмного забезпечення (наприклад, структури даних і розділення), так що специфікація, побудована у виконуваному UML, може бути розгорнута в різних програмних середовищах без змін. [37]. Ключовим інгредієнтом будь-якого виконуваного варіанту UML є використання мови дій (своєрідного псевдокоду), що дозволяє розробникам повністю вказувати детальні поведінкові аспекти моделі (наприклад, визначати поведінку методу класу).

Але навіщо нам ALF (Activity Language Framework) ? Такі мови програмування, як Java, C++ або інші мови програмування, не призначені для маніпулювання елементами CS. Вони не надають засобів, необхідних для того, щоб ми могли виражати дії в CS чітко й точно, але абстрактно. Однак мови програмування дозволяють розробнику маніпулювати всіма видами специфічних для реалізації функцій, які є абсолютно недоречними. Наприклад, у моделюванні є звичайним явищем переміщення через асоціацію (тобто пошук пов’язаних об’єктів на іншому кінці асоціації). З мовою програмування нам потрібно знати, як буде реалізовано асоціацію, наприклад, з будь-якою структурою даних, тому керуйтеся асоціацією, використовуючи операції, пов’язані з цією структурою даних. Це відразу робить платформу реалізації моделі специфічною. Однак ALF дозволяє просто та лаконічно керувати асоціаціями, не обмежуючи шляхи реалізації асоціацій.

ALF є незалежною від платформи мовою, яка працює на тому самому семантичному рівні, що й решта CS на основі UML. Це означає, що дії дозволяють пряме маніпулювання елементами PIM (не робиться жодних припущень щодо проміжного програмного забезпечення, мови реалізації чи політики проектування програмного забезпечення), і їх можна транслювати в різні реалізації для різних платформ і мов. Синтаксично ALF базується на кількох ключових принципах проектування [12]:

• ALF має переважно застарілий синтаксис C («подібний до Java»), оскільки він найбільш знайомий спільноті, яка програмує детальну

поведінку. Незважаючи на це, ALF допускає текстовий синтаксис UML, якщо він існує (наприклад, синтаксис двокрапки для введення тексту, синтаксис подвійної двокрапки для кваліфікації імені тощо).

• ALF не вимагає зміни графічних моделей, щоб забезпечити використання мови дій (наприклад, дозволені спеціальні символи в іменах, довільні імена дозволені для конструкторів тощо). Крім того, хоча ALF відображається на підмножині fUML, щоб забезпечити семантику виконання, її можна використовувати в контексті моделей, не обмежених підмножиною fUML.

• ALF використовує неявну систему типів, яка дозволяє, але не вимагає явного оголошення типізації в межах діяльності, завжди забезпечуючи перевірку статичного типу, засновану принаймні на типі, оголошеному в елементах структурної моделі.

• ALF має експресивність OCL у використанні та маніпулюванні послідовностями значень. Ці вирази послідовності повністю виконувані з точки зору областей розширення fUML, дозволяючи просту та природну специфікацію висококонкурентних обчислень.

• ALF надає систему імен, яка базується на просторах імен UML для посилань на елементи поза діяльністю, але також забезпечує послідовне використання локальних імен для посилань на потоки значень у діяльності. ALF додає концепцію одиниці до основних концепцій UML просторів імен і пакетів. Одиниця — це простір імен, визначений за допомогою нотації ALF, який сам по собі текстово не міститься в будь-якому іншому визначенні простору імен ALF. Одиниці — це лексично незалежні (хоча семантично пов’язані) сегменти тексту ALF. На рисунку 3.2 показано визначення одиниці ALF для цього прикладу. У цьому визначенні ми можемо побачити класи та асоціації, які утворюють пакет VideoClub.

Блок також може мати субодиниці, які визначають простори імен, якими володіє (прямо чи опосередковано) одиниця, але чиє визначення ALF надається одиницею, яка текстово відокремлена від базової одиниці.

Включення в базовий блок вказується за допомогою оголошення заглушки в базовому блокі та оголошення простору імен у визначенні субблока.

```
package VideoClub {            // Unit definition
    public class RentalLine;   // stub declaration
    public class VideoClub;
    public class Rental;
    public class Partner;
    public class Movie;

public assoc partner_rental;
public assoc movie_rentalline;
public assoc rental_rentalline;
public assoc videoclub_movie;
}
```

Рис. 3.2. Текстове визначення для пакета VideoClub за допомогою мови ALF

Таким чином, ми генеруємо CS під час тестування, використовуючи структурну частину (діаграма класів з попередніми, постумовами та інваріантами) і перетворюючи CSUT в одиниці ALF і перетворюючи попередні, постумови та інваріанти в поведінкову інформацію (тобто методи), яка буде використовуватися в Виконання CSUT (для тестування) (рис. 3.3).

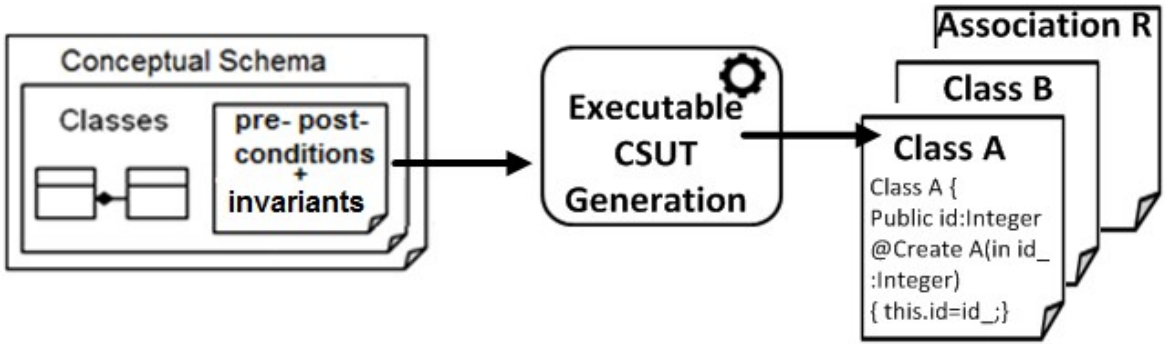

Рис. 3.3. Огляд створення виконуваного файлу

Ми вирішили використовувати еталонну реалізацію як механізм fUML, оскільки (1) вона базується на еталонній реалізації та (2) забезпечує журнал виконання. Завдяки (1) ми маємо впевненість у його відповідності

специфікації fUML. I (2) означає, що систематичне тестування (тобто перегляд сотень журналів) простіше, ніж з Moka реалізація, яка більше підходить для інтерактивного тестування.

Трансляція CS на основі UML CD у ALF виконується у два етапи:

1. Перетворення режиму в текст перетворює CS на основі UML CD в одиниці ALF. Це перетворення записане в коді ATL. Він приймає як вхідні дані CS на основі UML CD, а видає CS на основі ALF. Результуючий CS на основі Alf містить елементи, згенеровані з трансляції всіх елементів CS, наданих як вхідні дані.

2. Розбір блоку ALF. Семантично ALF відображає CS на підмножину фундаментального UML (fUML [67]). Отримана ALF-основа CS семантично еквівалентна вихідній. Потім fUML надає віртуальну машину для виконання модулів ALF.

Концептуальна схема на основі ALF можна запустити з командного рядка за допомогою сценарію оболонки ALF (для Unix) або пакетного файлу alf.bat (для Windows/DOS). CS на основі ALF компілюється в представленні в пам’яті та виконується за допомогою еталонної реалізації fUML. Подальшу інформацію можна знайти в еталонній реалізації ALF [17].

Поточна версія нашого перетворення ALF підтримує більшість конструкцій UML за такими помітними винятками:

1) функції, необхідні для визначення абстракцій, можуть бути додані з відносно невеликою роботою;

2) трансформація обмежень OCL.

Наразі концептуальна схема (CS) на основі UML CD, які використовуються в нашому підході, використовують безпосередньо мову ALF для визначення обмежень. Але існує підхід, що дозволяє інтегрувати OCL і fUML шляхом перетворення, який можна використати для вирішення цієї проблеми [11].

3.2. Архітектура та середовище тестування

Щоб виконати концептуальне тестування схеми, наша система перевірки базується на архітектурі, показаній на схемі на рисунку 3.4.

У нашій структурі перевірки концептуальних схем розробники концептуального моделювання визначають явну специфікацію концептуальної схеми інформаційної системи, що розробляється. Потім створюється набір автоматизованих тестів для перевірки схеми за допомогою нашої тестової системи. Формальна мова для визначення концептуальної схеми та формальна мова для визначення тестових програм необхідні, щоб зробити цей підхід застосовним на практиці. У цій роботі ми перевіряємо концептуальні схеми, визначені мовами UML і ALF.

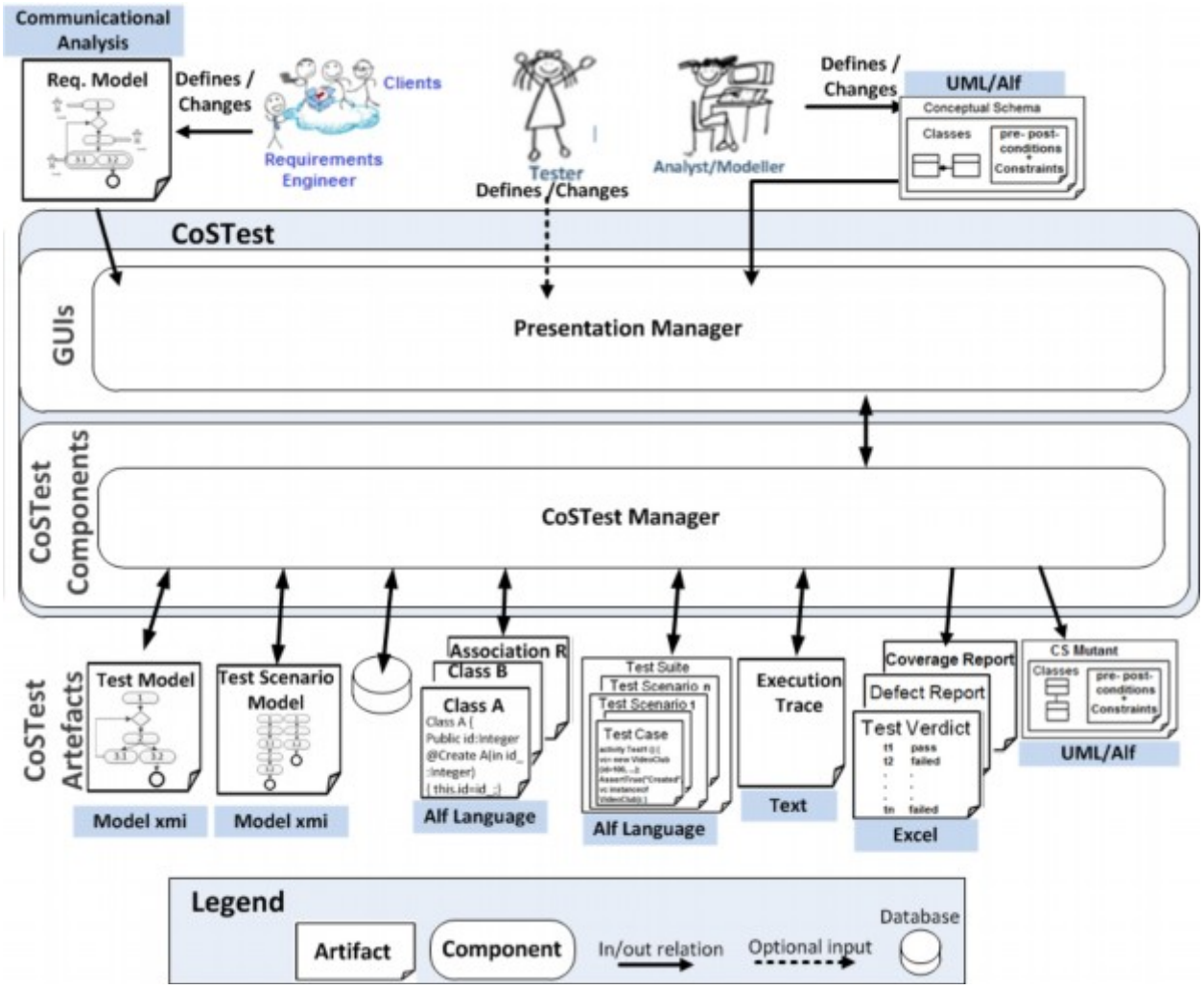

Рис. 3.4. Середовище тестування для валідації концептуальних схем

Семантично ALF відображає CS у підмножину Foundational UML (fUML), після чого fUML надає віртуальну машину для виконання мови ALF. CSUT на основі ALF можна виконати з командного рядка за допомогою сценарію оболонки ALF (для Unix) або пакетного файлу alf.bat (для Windows/DOS).

Виконання CSUT компілюється у внутрішнє подання пам'яті та виконується за допомогою еталонної реалізації fUML. Синтаксис використання:

```
alf [опції] назва_блоку
```

де назва_блоку - повне кваліфіковане ім'я блоку (наприклад, тестовий випадок), який потрібно виконати. Допустимі опції:

-d рівень: Встановлює рівень зневадження для виведення трасування з рушія виконання fUML. Корисні рівні:

OFF - вимикає виведення трасування.

ERROR - повідомляє лише про серйозні помилки (наприклад, коли під час виконання неможливо знайти реалізацію примітивної поведінки).

INFO - виводить основну інформацію про трасування виконання дій та активностей.

DEBUG - виводить детальну інформацію про трасування виконання активності.

-f: Розглядає назва_блоку як ім'я файлу, а не кваліфіковане ім'я. Очікується, що зазначений файл буде знайдено безпосередньо в каталозі моделі, а блок повинен мати таке саме ім'я, як і ім'я файлу (без розширення .alf).

-l шлях: Встановлює розташування каталогу бібліотек за адресою шлях. Якщо ця опція не вказана, а змінне середовище ALF_LIB встановлено, то значення ALF_LIB використовується як розташування каталогу бібліотек. В іншому випадку використовується значення за замовчуванням - Libraries.

-m шлях: Встановлює розташування каталогу моделі за адресою шлях. Визначення повного кваліфікованого імені для шляхів файлів блоку відносно кореня каталогу моделі. Якщо ця опція не вказана, використовується значення за замовчуванням - Models.

-p: Аналізує та перевіряє обмеження для визначеного блоку, але не виконує його. Це корисно для синтаксичної та статичної семантичної перевірки блоків, які самі по собі не є виконуваними.

-P: Аналізує та перевіряє обмеження, як для опції -p, а потім друкує отримане дерево абстрактного синтаксису. Зверніть увагу, що роздруківка відбудеться навіть за наявності порушень обмежень.

-v: Встановлює детальний режим, у якому друкуються повідомлення про стан щодо аналізу та інших етапів обробки, що ведуть до виконання. Якщо цю опцію використовується окремо, без зазначення назви блоку (тобто, alf -v), то друкується лише інформація про версію.

Детальнішу інформацію можна знайти у Wiki еталонної реалізації ALF. ALF також є такою мовою, але вона є стандартом OMG, яку можна послідовно реалізувати в ряді інструментів, що сприяє тій самій взаємодії для текстової поведінкової специфікації, яку вже робить стандарт UML для графічного моделювання. Саме тому в цій дисертації ми зосереджуємося на мові ALF як на нашому тестовому середовищі. Однак ідеї, представлені в цьому документі, можуть бути адаптовані до будь-якої з перерахованих мов дій.

```
Constraint violations:
  behaviorInvocationExpressionReferentConstraint          in
C:\Users\Usuario\workspace\COSTest\ExecutableTestCases\UML-
ALF\VideoClub\/VideoClub_TS_1_TC_36.alf at line 17, column 12
  instanceCreationExpressionDataTypeCompatibility         in
C:\Users\Usuario\workspace\COSTest\ExecutableTestCases\UML-
ALF\VideoClub\/VideoClub_TS_1_TC_36.alf at line 20, column 22

  positionalTupleArguments                                in
C:\Users\Usuario\workspace\COSTest\ExecutableTestCases\UML-
ALF\VideoClub\/VideoClub_TS_1_TC_36.alf at line 20, column 34
```

Рис. 3.5. Приклад трасування виконання для Video Club CS

Сліди виконання, отримані в результаті виконання тестових прикладів, налаштовані для звітування про помилки та синтаксичні помилки, виявлені під час процесу тестування аналізатором ALF. На рисунку 3.5 показано приклад трасування виконання для Video Club CS.

3.3. Оцінка тесту

Оскільки тести є частиною процесу перевірки та верифікації, автоматизовані процедури (тобто аналіз синтаксису та покриття) використовувалися для перевірки моделей як попереднього кроку до процесу тестування.

3.3.1. Перевірка правильності синтаксису

Усі мови мають синтаксис, тобто набір правил щодо того, як елементи мови можуть осмислено поєднуватися разом у цій мові. Тоді специфікації, написані на певній мові, повинні відповідати синтаксису, накладеному мовою, якою вони визначені. Цей зв’язок між специфікацією та мовою, якою вона описана, називається відповідністю.

Ми вважаємо виконувану концептуальну схему синтаксично правильною, якщо всі елементи задовольняють правила, визначені в метамоделі UML/fUML і правила правильної форми (WFR) - обмеження, які обмежують можливий набір дійсних (або добре сформованих) моделей.

Розглянемо уривок діаграми класів, показаної на рисунку 3.6 і операцію конструктора (в контексті класу CorporatePartner) для створення екземпляра цього класу.

Наведена вище операція не є синтаксично правильною, оскільки виклик альтернативного конструктора не є першим рядком у визначенні методу операції конструктора. Потім відредагована операція показана на рисунку 3.7.

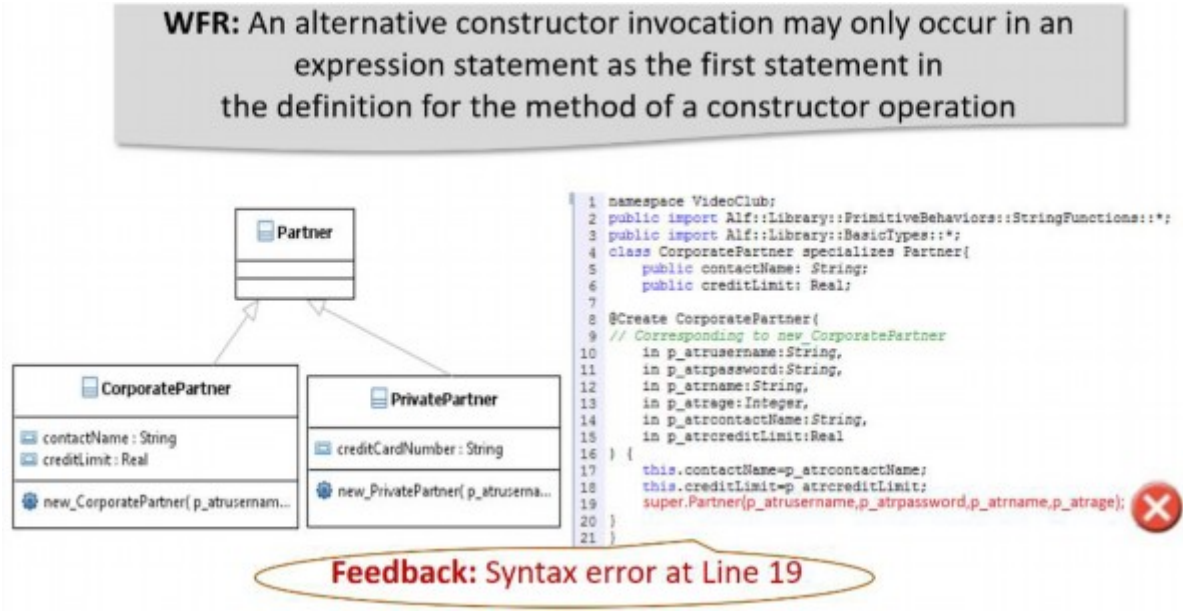

Рис. 3.6. Фрагмент концептуальної схеми із синтаксично неправильним кодом

```
1  namespace VideoClub;
2  public import Alf::Library::PrimitiveBehaviors::StringFunctions::*;
3  public import Alf::Library::BasicTypes::*;
4  class CorporatePartner specializes Partner{
5      public contactName: String;
6      public creditLimit: Real;
7
8  @Create CorporatePartner(
9  // Corresponding to new_CorporatePartner
10     in p_atrusername:String,
11     in p_atrpassword:String,
12     in p_atrname:String,
13     in p_atrage:Integer,
14     in p_atrcontactName:String,
15     in p_atrcreditLimit:Real
16 ) {
17     super.Partner(p_atrusername,p_atrpassword,p_atrname,p_atrage);
18     this.contactName=p_atrcontactName;
19     this.creditLimit=p_atrcreditLimit;
20 }
21 }
```

Рис. 3.7. Приклад концептуальної схеми з виправленим ALF кодом

3.3.2. Перевірка семантичної правильності

Ми вважаємо виконувану концептуальну схему (тобто набір одиниць ALF з операціями на основі дій) семантично правильною, якщо всі можливі зміни (вставки/оновлення/видалення/...) у всіх частинах стану системи можна виконати через виконання цих операцій. Елемент існує, але деякі твердження

щодо домену є неправильними. Наприклад, розглянемо уривок діаграми класів і тестовий приклад, складений за допомогою операції set_status, показаної на рисунку 3.8.

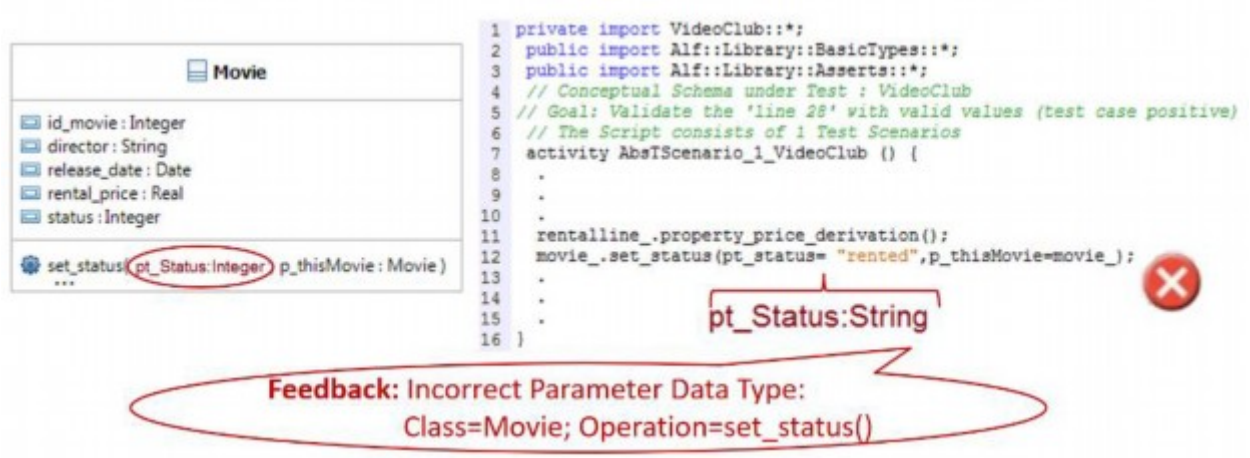

Рис. 3.8. Фрагмент концептуальної схеми VideoClub із семантично некоректним дефектом

Ця концептуальна схема є семантично неправильною, оскільки, наприклад, операція set_status існує, але очікуваний тип параметра (тобто String) відрізняється від очікуваного (тобто Integer). Потім, щоб виправити цю семантичну помилку, дизайнер повинен змінити тип параметра pt_Status на Integer. Відлагоджена операція показана на рисунку 3.9.

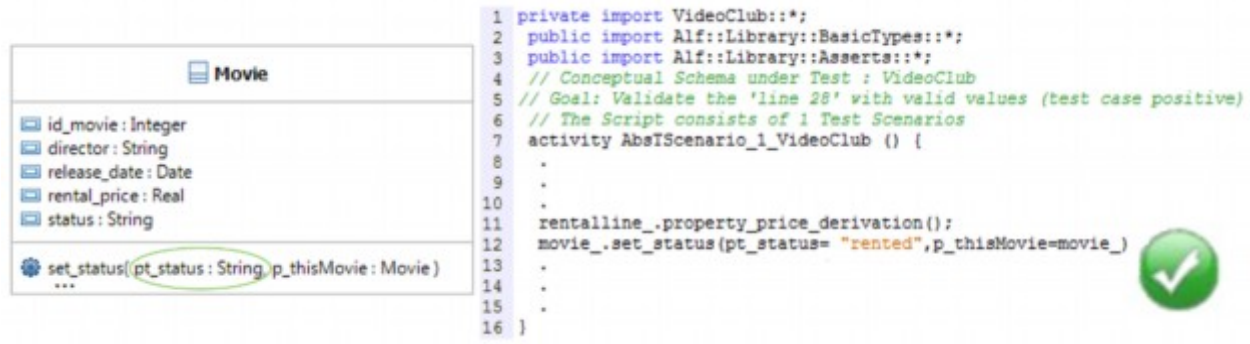

Рис. 3.9. Приклад концептуальної схеми VideoClub з виправленим семантичним дефектом

3.3.3. Перевірка непотрібних елементів

Крім того, непотрібні елементи (тобто зайві/повторювані елементи або сторонні елементи) у схемі можна виявити шляхом аналізу охоплення елементів, включених до концептуальної схеми та виконані в тестових випадках. Приклад CS, що містить сторонню асоціацію, показано на рисунку 3.10.

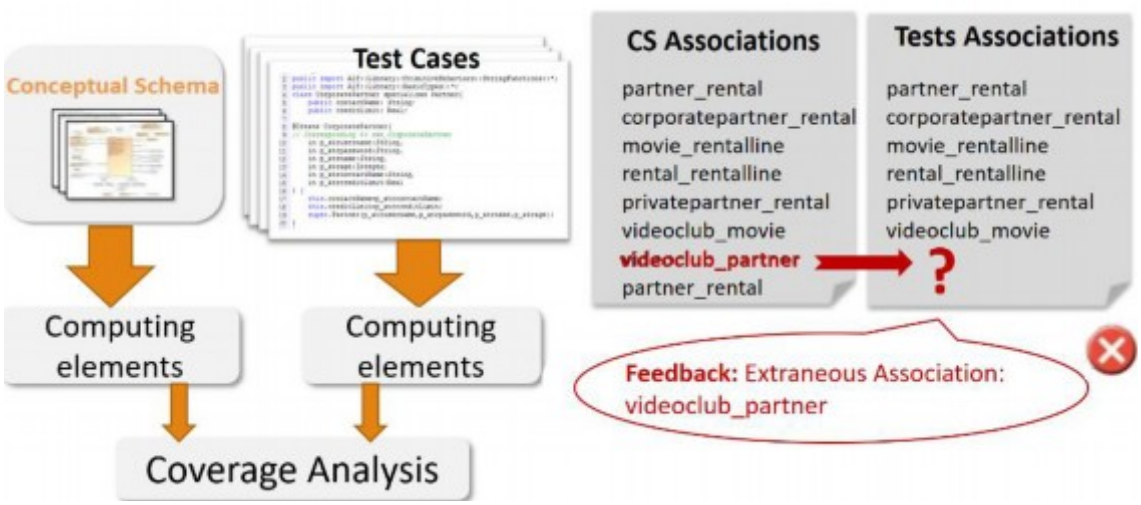

Рис. 3.10. Приклад концептуальної схеми, що містить сторонню асоціацію

3.3.4. Перевірка повноти

Метод, який ми розробили для перевірки властивості повноти, приймає як вхідні дані виконувану модель, що складається зі структурної моделі (діаграма класів UML) і моделі поведінки (набір операцій Alf). Таким чином, концептуальна схема є завершеною, якщо всі елементи, що застосовуються в тестових випадках, існують на CS.

Тоді наш метод повертає або позитивну відповідь, що означає, що модель поведінки завершена, або коригувальний зворотний зв’язок, що складається з набору дій, які повинні бути включені в певну операцію поведінкової моделі, щоб зробити її повною.

Наприклад, розглянемо фрагмент діаграми класів і тестовий приклад, створений за допомогою операції new Partner, показаної на рис. 3.11.

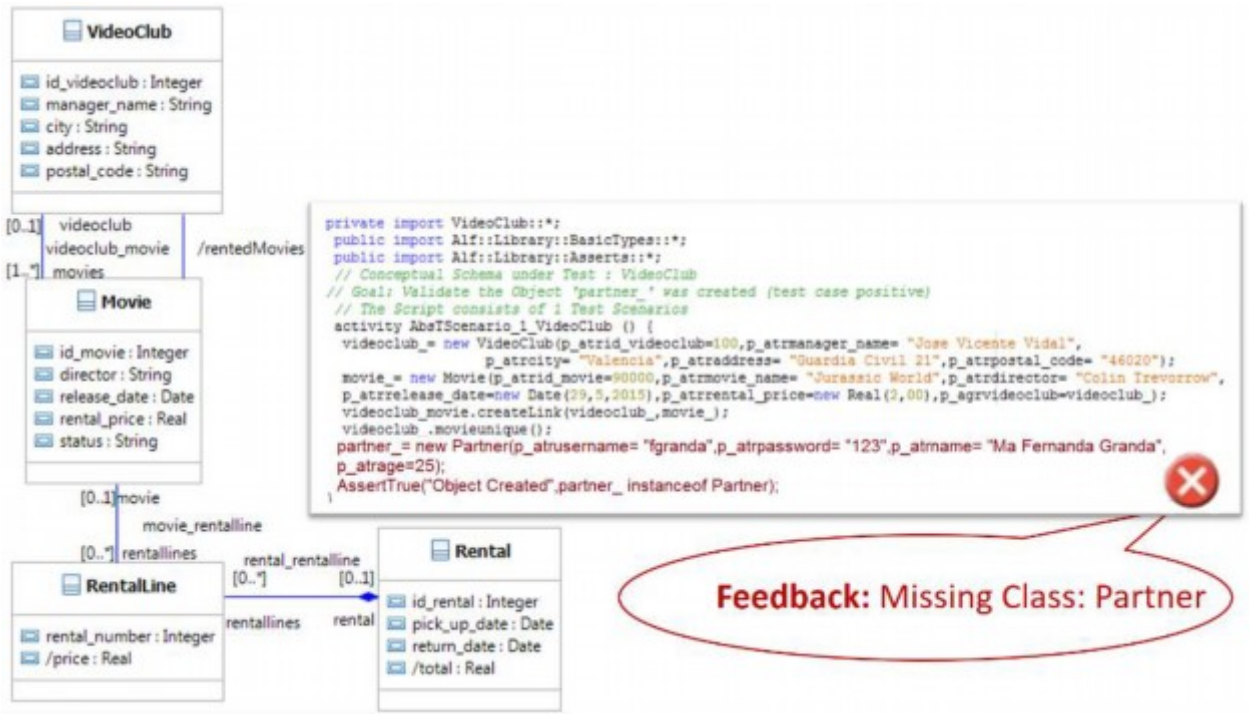

Рис. 3.11. Фрагмент концептуальної схеми з відсутнім дефектом

Ця концептуальна схема є неповною, оскільки, наприклад, клас Partner не існує. Потім, щоб виправити цей недолік, дизайнер повинен змінити CS, додавши клас Partner. Відкорегована операція показана на рис. 3.12.

Рис. 3.12. Фрагмент коректної концептуальної схеми

Цей етап виконується за допомогою оракулів і цілей, включених до тестових випадків. Тестовий приклад повертає вердикт «Пройшов», «Не пройшов» або «Непереконливо». Якщо вердикт невдало, надається список дефектів і статус невдалого виконання. Виконання тестів може призвести до виведення кількох дефектів (наприклад, відсутній клас, неправильна операція та відсутня операція), які містяться у списку. Коли вердикт є Непереконливим, це означає, що виконання тесту не є остаточний. Наприклад, якщо прилад спричинив несправність, це призводить до непереконливого статусу. Цей вердикт також може повертати список дефектів. В іншому випадку статус тестового прикладу буде проходити. Як приклад знову розглянемо концептуальну схему. Розробник/тестер концептуального моделювання, який бажає перевірити цю сутність сеансу, може виконати тестовий приклад, показаний на рисунку 3.13.

```
private import VideoClub::*;
 public import Alf::Library::BasicTypes::*;
 public import Alf::Library::Asserts::*;
 // Conceptual Schema under Test : PA_login
// Goal: Validate the Object 'session_' was created (test case positive)
 // The Script consists of 1 Test Scenarios
 activity AbsTScenario_1_PA_login () {
    partner_= new Partner(p_atrusername= "username",p_atrpassword= "clave2016",p_atrname= "Usuario Example",
              p_atrage= 19);
    session_= new Session(p_atrid_session=1,p_atrlogin_date=new Date(10,5,2016),p_atrlogin_time= "15:04",
              p_atrusername= "mfgranda");
    AssertTrue("Object Created",session_ instanceof Session);
}
```

Рис. 3.13. Приклад тесту

Після виконання тесту згенерований журнал помилок виглядає наступним чином (рис. 3.14).

```
--------Test Case: 2--------
Constraint violations:
  instanceCreationExpressionConstructor                                    in
C:\Users\Usuario\workspace\COSTest\ExecutableTestCases\UML-
ALF\VideoClub\/PA_login_TS_1_TC_2.alf at line 10, column 19
  positionalTupleArguments                                                 in
C:\Users\Usuario\workspace\COSTest\ExecutableTestCases\UML-
ALF\VideoClub\/PA_login_TS_1_TC_2.alf at line 10, column 31
  classificationExpressionTypeName                                         in
C:\Users\Usuario\workspace\COSTest\ExecutableTestCases\UML-
ALF\VideoClub\/PA_login_TS_1_TC_2.alf at line 12, column 37
```

Рис. 3.14. Приклад трасування виконання

Вердикт твердження – невдача. Потім трасування виконання аналізується за допомогою інформації, наведеної в таблиці 3.1. Потім повідомляється про відсутність класу (або приватного) дефекту. Тест буде пройдено, якщо схему буде виправлено, як показано на рисунку 3.15.

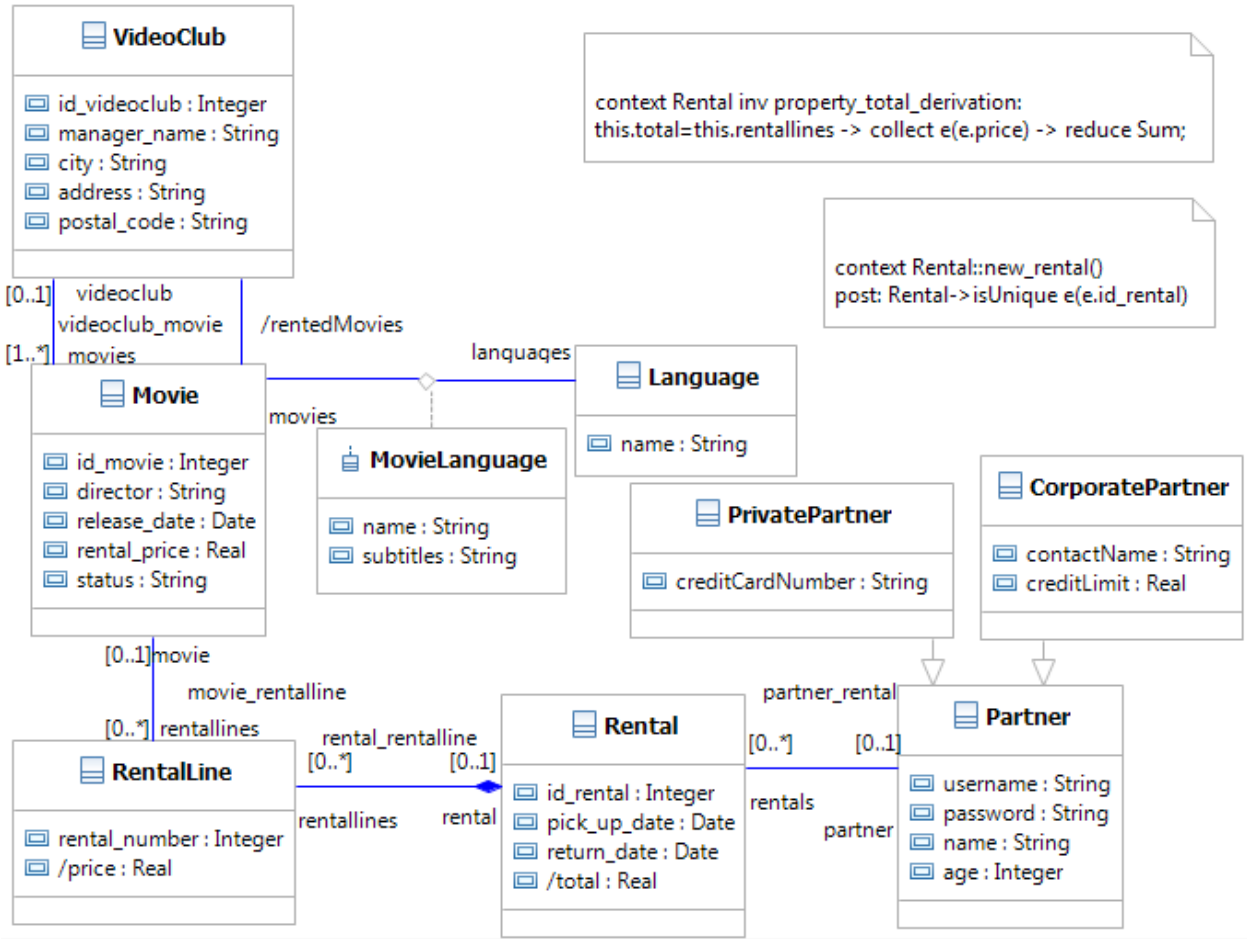

Рис. 3.15. Розширена діаграма класів UML для концептуальної схеми Video Club

Таблиця 3.1.

Зв'язок між несправністю та повідомленим дефектом

Fault reported by	Defect Reported
propertyAccessExpressionFeatureResolution	Missing or private Association
instanceCreationExpressionConstructor	Missing Class (or private)
behaviorInvocationExpressionReferentConstraint	Missing Operation (or private)

propertyAccessExpressionFeatureResolution	Incorrect Association
linkOperationExpressionArgumentCompatibility	Incorrect Association Ends
instanceCreationExpressionConstructorlessLegality	Incorrect Constructor
assignmentExpressionSimpleAssignmentTypeConformance	Incorrect Parameter Data Type
tupleNullInput in a createlink statement	Incorrect null Value in Association Parameter
tupleNullInput in an operation statement	Incorrect null Value in Parameter
instanceCreationExpressionDataTypeCompatibility	Incorrect Operation Signature
behaviorInvocationExpressionArgumentCompatibility	Incorrect Parameter Data Type
superInvocationExpressionOperation	Incorrect Super Class

Нарешті, оцінка тесту генерує звіт із вердиктами тестових випадків, виявленими помилками, звітом про час і охопленням тестових випадків.

3.4. Огляд процесу тестування концептуальних схем

Методи тестування для концептуальних схем UML, ймовірно, відрізнятимуться залежно від використовуваних критеріїв тестування [18]. Щоб проілюструвати процес тестування та висвітлити деякі проблеми, які необхідно було вирішити під час розробки, на рисунку 3.16 узагальнено процес тестування, який поділено на три етапи.

1. Створення набору тестів

1.1. Перетворення моделі вимог (на основі комунікаційного аналізу) у тестову модель.

1.2. Перетворення тестової моделі на модель тестового сценарію (послідовність подій із тестової моделі).

1.3. Згенерувати тестові значення для тестових випадків із тестової моделі (конкретизувати змінні). Тестер (опціонально) може вводити нові тестові значення.

1.4. Перетворити кожен тестовий сценарій у сценарії тестових випадків (сценарій ALF), який містить абстрактні тестові випадки.

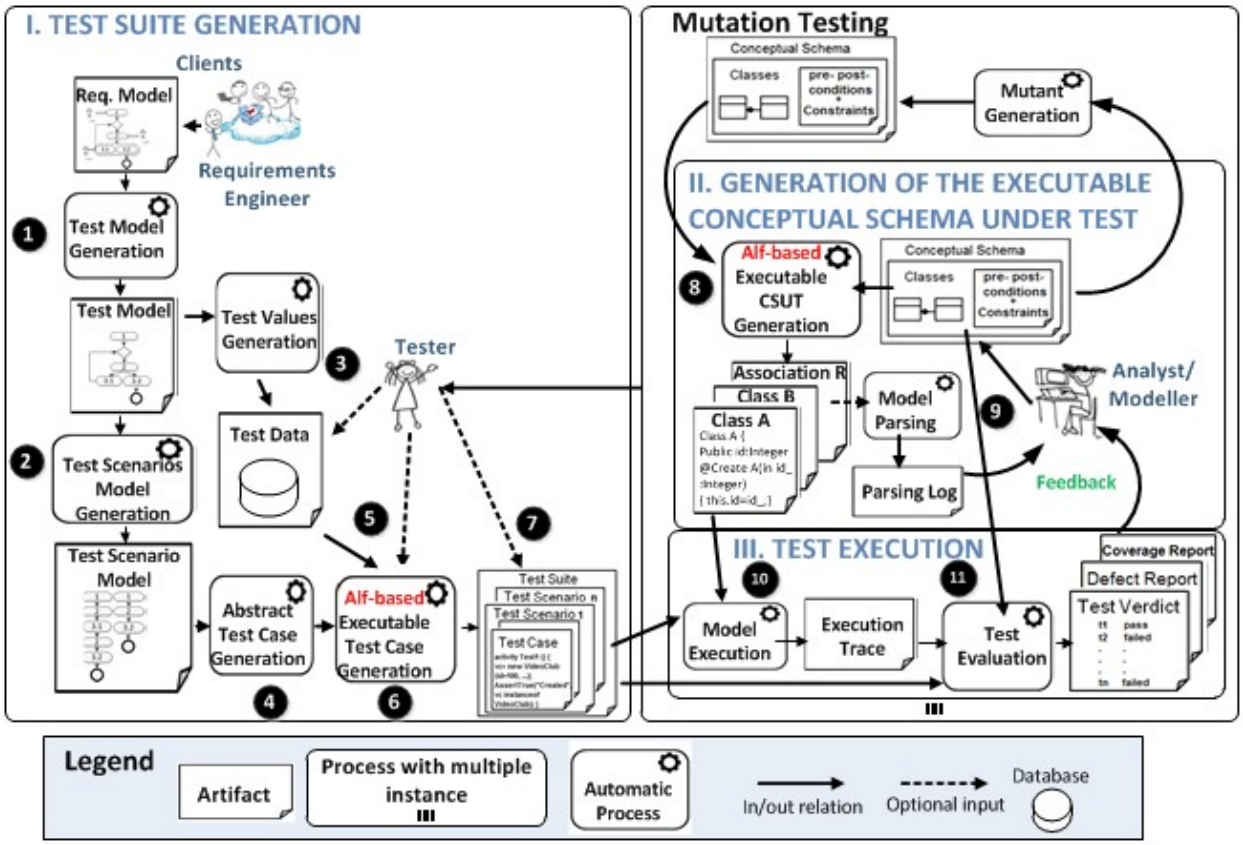

Рис. 3.16. Огляд процесу тестування концептуальних схем

1.5. Вибрати тип тестів (наприклад, лише позитивні тести або включно з негативними тестами).

1.6. Створити конкретні і виконувані тести.

1.7. Розставити пріоритети та вибрати тестові випадки для виконання на основі аналізу мутацій.

2. Генерація виконуваної концептуальної схеми під час тестування.

2.1. Створити CS, що тестується, використовуючи як структурну (діаграма класів), так і поведінкову інформацію (передумови, постумови та інваріанти), а також перетворити CSUT в одиниці ALF для використання у виконанні моделі (цілі тестування).

2.2. Проаналізувати CS перед початком виконання CS (процес тестування).

3. Виконати тест.

3.1. Виконати тестові випадки (скрипти) проти CS, що тестується.

3.2. Створити звіт про тестування та аналіз покриття.

Отже, наша пропозиція щодо створення тестових випадків відповідає принципам тестування на основі моделі. Ми пропонуємо перевірку концептуальних схем на основі тестування, щоб посилити перевірку повноти (відсутні елементи). Однак обмеження та змінність можна покращити шляхом аналізу охоплених і не охоплених елементів (зайвих елементів) за допомогою тестових випадків. Оскільки процес тестування має перетворити CSUT у виконуваний формат, надлишкові та неправильні елементи виявляються синтаксичним аналізатором як попередній крок до тестування, так що мета правильності також покращується.

Процес тестування базується на тестових сценаріях для виконання концептуальних схем високого рівня незалежно від платформи за допомогою стандартної мови дій від OMG, ALF. Структура перевірки використовує низхідний підхід для створення тестових випадків, де тестова модель є головною, яка генерує тестові сценарії та тестові випадки. Щоб автоматизувати створення набору тестів, ми вибрали керовану моделлю архітектуру для аналізу, проектування та впровадження. Тому дизайн тесту не залежить від рівня адаптації або системи виконання тесту, а артефакти тесту не залежать від домену реалізації.

Висновки до розділу

Третій розділ зосереджений на практичних аспектах імплементації моделей і засобів для валідації концептуальних схем у модельно-орієнтованому середовищі. У ньому розглянуто, як виконавчі концептуальні схеми, засновані на UML, інтегруються в архітектуру тестування, а також оцінено їхню якість за допомогою комплексного тестування.

Діаграми класів UML розглядаються як основа для побудови виконуваних концептуальних схем. Ці схеми дозволяють не лише візуалізувати структуру системи, а й виконувати перевірки в реальному часі,

що забезпечує більш динамічний підхід до валідації. Такий підхід сприяє виявленню логічних і семантичних помилок ще до етапу реалізації.

Ефективна імплементація моделей валідації передбачає створення архітектури, яка інтегрує інструменти для тестування та управління процесами. Середовище тестування, орієнтоване на моделі, забезпечує автоматизацію багатьох етапів перевірки, спрощуючи процес і підвищуючи його точність.

Процес тестування концептуальних схем описано як поетапний підхід, який включає підготовку, виконання тестів та аналіз результатів. Особливу увагу приділено інтеграції автоматизованих інструментів для перевірки, що дозволяє скоротити час на тестування та підвищити його ефективність.

У підсумку, імплементація моделей та засобів валідації концептуальних схем у модельно-орієнтованому середовищі забезпечує більш точне, гнучке та ефективне тестування. Використання UML як основи дозволяє досягти високого рівня стандартизації та автоматизації, що сприяє створенню якісних програмних рішень.

ВИСНОВКИ

У магістерській роботі досліджено концептуальні схеми як фундаментальний інструмент забезпечення якості програмного забезпечення, а також методи їхньої валідації на основі процесів тестування в модельно-орієнтованому середовищі. Результати дослідження підтвердили важливість комплексного підходу до побудови, перевірки та вдосконалення концептуальних моделей, що дозволяє оптимізувати процеси розробки та мінімізувати помилки.

Концептуальні схеми є основою для опису вимог, структури та поведінки програмного забезпечення. Їхня якість визначає успіх процесів верифікації та валідації. Використання стандартів, таких як UML, забезпечує уніфікованість і зрозумілість моделей.

У роботі розроблено підхід до валідації на основі тестування, який включає етапи від моделювання вимог до генерації тестів. Застосування аналітичних методів, таких як комунікаційний аналіз, дозволяє детально формулювати вимоги та перевіряти їх відповідність.

Виконувані концептуальні схеми забезпечують можливість перевірки моделей у реальному часі, що дозволяє знаходити логічні, семантичні та синтаксичні помилки на ранніх етапах розробки. Автоматизація тестування в модельно-орієнтованому середовищі знижує трудомісткість і підвищує точність перевірки.

Запропоновані методи тестування охоплюють перевірку правильності, повноти, актуальності та відповідності вимогам. Використання критеріїв якості дозволяє оцінювати ефективність концептуальних схем та їх придатність для подальшої реалізації.

Розроблена архітектура тестового середовища демонструє можливість інтеграції методів валідації концептуальних схем у сучасні процеси розробки. Вона підтримує автоматизацію генерації тестів, їх виконання та аналіз результатів.

Робота вносить практичний і теоретичний внесок у вдосконалення процесів розробки програмного забезпечення, зокрема через підвищення якості концептуальних моделей. Це дозволяє скоротити час і витрати на створення програмних продуктів, підвищуючи їх надійність і відповідність потребам замовників.

Загалом, магістерська робота показує, що поєднання стандартизованих підходів (UML), аналітичних методів та автоматизованих засобів тестування є ефективним інструментом для забезпечення високої якості програмних рішень.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. A. Olivé, Conceptual Modeling of Information System. Springer, 2007.
2. J. Johnson and A. Henderson, Conceptual Models: Core to Good Design. Morgan & Claypool, 2012.
3. R. France and B. Rumpe, “Model-driven Development of Complex Software: A Research Roadmap,” in International Conference on Software Engineering, 2007, no. 2, pp. 37–54.
4. M. Staron, “Adopting Model Driven Software Development in Industry – A Case Study at Two Companies,” Model Driven Eng. Lang. Syst., pp. 57–72, 2006.
5. J. Hutchinson, M. Rouncefield, and J. Whittle, “Model-driven Engineering Practices in Industry,” Proc. 33rd Int. Conf. Softw. Eng., pp. 633–642, 2011.
6. O. Pastor and J. C. Molina, Model-Driven Architecture in Practice. Cambridge: Springer Berlin Heidelberg, 2007.
7. R. Van Der Straeten and T. Mens, “Challenges in model-driven software engineering,” in Model Driven Engineering Languages and Systems - MODELS 2008, 2009, pp. 35–47.
8. A. Olivé and J. Cabot, “A Research Agenda for Conceptual Schema- Centric Development,” in Conceptual Modelling in Information Systems Engineering, 2007, pp. 319–334.
9. P. Mohagheghi, V. Dehlen, and T. Neple, “Definitions and approaches to model quality in model-based software development - A review of literature,” Inf. Softw. Technol., vol. 51, no. 12, pp. 1646–1669, 2009.
10. B. Unhelkar, Verification and Validation for Quality of UML 2.0 Models. WILEY, 2005.
11. P. Skoković and M. Rakić-Skoković, “Requirements-Based Testing Process in Practice,” vol. 1, no. 4, pp. 155–161, 2010.
12. P. Loucopoulos and V. Karakostas, System Requirements Engineering. McGraw-Hill Publishing Company, 1995.

13. S. España, A. González, and Ó. Pastor, “Communication Analysis: A Requirements Engineering Method for Information Systems,” in 21st International Conference on Advanced Information Systems Engineering, 2009, vol. 5565, pp. 530–545.
14. G. H. Travassos, F. Shull, and J. Carver, “Working with UML: A Software Design Process Based on Inspections for the Unified Modeling Language,” Adv. Comput., vol. 54, pp. 35–98, 2001.
15. Beizer, B. Software Testing Techniques. Van Nostrand Reinhold, 1990.
16. Myers, G. J., Sandler, C., Badgett, T. The Art of Software Testing. Wiley, 2011.
17. Binder, R. V. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley, 1999.
18. Sommerville, I. Software Engineering. Addison-Wesley, 2015.
19. Lutz, R. R. Software Engineering for Safety: A Roadmap. ACM, 2000.
20. Harel, D. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 1987.
21. Pressman, R. S. Software Engineering: A Practitioner's Approach. McGraw-Hill, 2020.
22. Fowler, M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, 2003.
23. Rumbaugh, J., Blaha, M., Premerlani, W. Object-Oriented Modeling and Design. Prentice Hall, 1991.
24. Booch, G. Object-Oriented Analysis and Design with Applications. Addison-Wesley, 2007.
25. Bertolino, A. Software Testing Research: Achievements, Challenges, Dreams. ACM, 2007.
26. Clements, P., Kazman, R., Klein, M. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley, 2002.
27. Offutt, J., Abdurazik, A. Generating Tests from UML Specifications. ACM, 1999.

28. Marick, B. The Craft of Software Testing. Prentice Hall, 1995.
29. Whittaker, J. A. How to Break Software: A Practical Guide to Testing. Addison-Wesley, 2002.
30. Dijkstra, E. W. A Discipline of Programming. Prentice Hall, 1976.
31. McGraw, G. Software Security: Building Security In. Addison-Wesley, 2006.
32. Kaner, C., Falk, J., Nguyen, H. Q. Testing Computer Software. Wiley, 1999.
33. Kitchenham, B. Procedures for Performing Systematic Reviews. Keele University, 2004.
34. ISO/IEC 25010:2011. Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE).
35. Perry, D. E., Wolf, A. L. Foundations for the Study of Software Architecture. ACM, 1992.
36. Zave, P., Jackson, M. Four Dark Corners of Requirements Engineering. ACM Transactions on Software Engineering and Methodology, 1997.
37. DeMillo, R. A., Lipton, R. J., Sayward, F. G. Hints on Test Data Selection: Help for the Practicing Programmer. IEEE, 1978.
38. Bach, J. Exploratory Testing Explained. STAREAST, 2003.
39. Parnas, D. L., Clements, P. C. A Rational Design Process: How and Why to Fake It. IEEE, 1986.
40. Bertolino, A., Marchetti, E. Introducing a Reference Framework for Testing UML. ACM, 2004.
41. Basili, V. R., Caldiera, G., Rombach, H. D. The Goal Question Metric Approach. Wiley, 1994.
42. Craig, R. D., Jaskiel, S. P. Systematic Software Testing. Artech House, 2002.
43. IEEE 829-2008. Standard for Software and System Test Documentation.
44. Mathur, A. P. Foundations of Software Testing. Addison-Wesley, 2008.
45. McConnell, S. Code Complete: A Practical Handbook of Software Construction. Microsoft Press, 2004.

46. Mens, T., Demeyer, S. Software Evolution. Springer, 2008.
47. Jalote, P. An Integrated Approach to Software Engineering. Springer, 2010.
48. Pooley, R., Stevens, P. Using UML: Software Engineering with Objects and Components. Addison-Wesley, 2006.
49. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
50. M. E. Fagan, “Design and code inspections to reduce errors in program development,” IBM Syst. J., vol. 15, no. 3, pp. 182–211, 1976.
51. C. Rolland and C. Proix, “A Natural Language Approach for Requirements Engineering,” in The 5th International Conference on Advanced Information Systems Engineering (CAiSE’93), 1993, pp. 257–277.
52. M. F. Granda, N. Condori-fernández, T. E. J. Vos, and O. Pastor, “What do we know about the Defect Types detected in Conceptual Models ?,” in IEEE 9th Int. Conference on Research Challenges in Information Science (RCIS), 2015, pp. 96–107.
53. A. Van Lamsweerde, Requirements Engineering: From System Goals to UML Models to Software Specifications. 2009.
54. O. I. Lindland, G. Sindre, and A. Sølvberg, “Understanding Quality in Conceptual Modeling,” IEEE Softw., vol. 11, no. 2, pp. 42–49, 1994.
55. T. Dinh-Trong, N. Kawane, S. Ghosh, and R. France, “A toolsupported approach to testing UML design models,” in 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS’05), 2005.