МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 15.00.00.000 ПЗ

Група ІПм-23-2

Маковійчук Максим

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Маковійчук Максим Юрійович

(прізвище, ім'я, по батькові)

УДК 004.94

(індекс)

МАГІСТЕРСЬКА РОБОТА

Методологія формальних специфікацій процесу генерації коду

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Маковійчук М.Ю.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Крихівський Михайло Васильович, к.т.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

Маковійчуку Максиму Юрійовичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Методологія формальних специфікацій процесу генерації коду”

керівник проекту (роботи) Крихівський Михайло Васильович, к.т.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 генерації коду

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

1. Теоретичний огляд передумов моделювання процесів генерації коду
2. Дослідження статичної та динамічної семантики для процесів моделювання ПЗ
3. Дослідження інструментів формальної специфікації
4. Представлення моделей та методології формальних специфікацій процесу генерації коду

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

1. Алгоритм тестування на основі моделі (рис. 1.1)
2. Робочий процес аналізу моделей (рис. 1.2)
3. Приклад моделі даних, розробленої за допомогою ASOME (рис. 1.3)
4. Приклад коду моделі даних (рис. 1.4)
5. Специфікація служби сховища (RS) (рис. 2.1)

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	Теоретичний огляд передумов моделювання процесів генерації коду	15.10.2024	виконано
4	Дослідження статичної та динамічної семантики для процесів моделювання ПЗ	08.11.2024	виконано
5	Дослідження інструментів формальної специфікації	20.11.2024	виконано
6	Представлення моделей та методології формальних специфікацій процесу генерації коду	01.12.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	13.12.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 82 с., 26 рис., 53 джерела.

Тема: Методологія формальних специфікацій процесу генерації коду

Об'єкт дослідження: процес розробки та генерації коду програмного забезпечення з використанням доменно-специфічних мов моделювання.

Мета роботи: розробка та верифікація методології формальних специфікацій процесу генерації коду, яка базується на доменно-специфічних моделях, для підвищення узгодженості та коректності процесу розробки програмного забезпечення.

Предмет дослідження: методи та методологія формальних специфікацій процесу генерації коду за допомогою доменно-специфічної мови DMDSL та інструментів верифікації.

Результати дослідження

В роботі проведена розробка формальної методології генерації коду на основі доменно-специфічних моделей з використанням мови Alloy. Запропонований підхід дозволяє перевіряти коректність виконання CRUD операцій і забезпечувати узгодженість моделі з вимогами системи.

Висновок

Результати проведеного дослідження підтвердили, що використання формальних методів, зокрема мови Alloy, для опису процесу генерації коду дозволяє покращити надійність і точність моделей, що використовуються в сучасних програмних системах.

ФОРМАЛЬНІ СПЕЦИФІКАЦІЇ, ВЕРИФІКАЦІЯ, МОДЕЛЮВАННЯ ПРОЦЕСІВ, ГЕНЕРАЦІЯ КОДУ, СТАТИЧНА ТА ДИНАМІЧНА СЕМАНТИКА, УЗГОДЖЕНІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ABSTRACT

Master Thesis: 82 pp., 26 fig., 53 sources.

Thesis Subject: Methodology of formal specifications of the code generation process

Research object: the process of developing and generating software code using domain-specific modeling languages.

The purpose of the work: development and verification of the methodology of formal specifications of the code generation process, which is based on domain-specific models, to increase the convenience and correctness of the software development process.

Research subject: methods and methodology of formal specifications of the code generation process using domain-specific language DMDSL and verification tools.

Research results

The paper developed a formal code generation methodology based on domain-specific models using the Alloy language. The proposed approach allows you to check the correctness of CRUD operations and ensure the consistency of the model with the system requirements.

Conclusion

The results of the research confirmed that the use of formal methods, in particular the Alloy language, to describe the code generation process allows to increase the reliability and accuracy of the models used in modern software systems.

FORMAL SPECIFICATIONS, VERIFICATION, PROCESS SIMULATION, CODE GENERATION, STATIC AND DYNAMIC SEMANTICS, SOFTWARE CONFORMITY

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API - Application Programming Interface

ASOME ASML - Software Modeling Environment

DMDSL - Domain-Interface Modeling Domain Specific Language.

MBT - Model-Based Testing.

OOP - Object-Oriented Programming.

SUT/IUT - System Under Test/Implementation

GPL - general-purpose language

RR - Repository Realization

RS - Service Specification

SCD - Source Cascade Delete

SIRE - Separating Interface and Realization

ВСТУП

Актуальність теми.

У сучасному світі розробка програмного забезпечення стає все більш складною та вимогливою до якості, надійності та безпеки. Застосування формальних методів у процесі моделювання та генерації коду дозволяє забезпечити узгодженість та коректність програмних систем на ранніх етапах розробки. Методи формальних специфікацій, зокрема використання доменно-специфічних мов, таких як DMDSL, і інструментів для формальної верифікації, як-от Alloy, надають можливість підвищити ефективність та якість процесу генерації коду. Актуальність теми полягає в необхідності розробки та впровадження таких підходів для забезпечення надійності програмного забезпечення в умовах зростаючої складності програмних систем.

Сучасні процеси розробки програмного забезпечення вимагають високих стандартів якості, безпеки та продуктивності. Зростаюча складність систем, інтеграція різноманітних функціональних компонентів та необхідність постійного вдосконалення розробницьких процесів призводять до того, що традиційні методи тестування та верифікації можуть бути недостатньо ефективними для забезпечення коректності та узгодженості програмних рішень. У такому контексті використання формальних методів моделювання і специфікації набуває особливої актуальності.

Однією з ключових проблем, яку вирішує дане дослідження, є підвищення рівня автоматизації та точності процесу генерації коду. Формальні методи, такі як доменно-специфічні мови (Domain-Specific Modeling Languages, DSL) і інструменти для формальної верифікації (наприклад, Alloy), забезпечують строгий підхід до перевірки узгодженості та правильності моделей на ранніх етапах розробки, що дозволяє уникнути багатьох проблем в подальшій реалізації програмних рішень.

У контексті розробки складних програмних систем та використання сучасних інструментів, таких як DMDSL (Domain-Modeling Domain-Specific Language) і ASOME (Advanced Software Modeling Environment), особливу увагу приділяють не тільки статичному моделюванню, але й динамічним аспектам системи, які забезпечують коректну роботу під час виконання програм. Актуальність цього дослідження також підкреслюється зростанням попиту на системи, які не тільки моделюють і перевіряють структури даних та операції, але й підтримують узгодженість виконання динамічних операцій, таких як створення, оновлення та видалення об'єктів.

З огляду на те, що автоматизація процесів генерації коду значно зменшує ризики людських помилок і підвищує продуктивність розробки, застосування формальних методів в цьому процесі є важливим кроком до побудови більш надійних і масштабованих систем. Це особливо актуально в критично важливих сферах, таких як банківська система, медицина, аерокосмічна галузь, де навіть найменші помилки в програмному забезпеченні можуть призвести до значних збитків або катастрофічних наслідків.

Отже, актуальність даного дослідження полягає в розробці та впровадженні формальних методів специфікації та верифікації процесу генерації коду для підвищення якості, надійності та продуктивності програмного забезпечення, що особливо важливо в умовах зростаючих вимог до складних систем та їхньої інтеграції.

Мета дослідження - розробка та верифікація методології формальних специфікацій процесу генерації коду, яка базується на доменно-специфічних моделях, для підвищення узгодженості та коректності процесу розробки програмного забезпечення.

Об’єкт дослідження - процес розробки та генерації коду програмного забезпечення з використанням доменно-специфічних мов моделювання.

Предмет дослідження - методи та методологія формальних специфікацій процесу генерації коду за допомогою доменно-специфічної мови DMDSL та інструментів верифікації.

Відповідно до мети роботи було сформовано наступні **задачі:**

- Провести огляд предметної області та методологій формального моделювання процесу генерації коду.
- Дослідити статичну та динамічну семантику доменно-специфічної мови DMDSL для моделювання процесів генерації коду.
- Розробити та впровадити формальні специфікації процесу генерації коду з використанням мови Alloy.
- Провести верифікацію та тестування моделей для перевірки узгодженості та коректності операцій CRUD+A.
- Оцінити ефективність застосування методів формальних специфікацій для підвищення якості та надійності програмного забезпечення.

Методи дослідження.

- Теоретичний аналіз існуючих методологій моделювання та специфікації програмного забезпечення.

- Моделювання процесів за допомогою доменно-специфічної мови DMDSL.

- Формальна верифікація специфікацій з використанням мови Alloy.

- Експериментальне тестування розроблених моделей на узгодженість та коректність.

Наукова новизна отриманих результатів полягає у розробці та верифікації формальної методології генерації коду на основі доменно-специфічних моделей з використанням мови Alloy. Запропонований підхід дозволяє перевіряти коректність виконання CRUD операцій і забезпечувати узгодженість моделі з вимогами системи.

Практичне значення магістерської роботи полягає в можливості застосування запропонованої методології для покращення процесів розробки

та тестування програмного забезпечення. Використання формальних специфікацій забезпечує високу якість та узгодженість програмних систем, що дозволяє зменшити кількість помилок та підвищити ефективність розробки.

Структура магістерської роботи. Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 82 сторінки, і містить 26 рисунків, список використаних джерел із 53 найменувань.

РОЗДІЛ 1. ТЕОРЕТИЧНИЙ ОГЛЯД ПЕРЕДУМОВ МОДЕЛЮВАННЯ ПРОЦЕСІВ ГЕНЕРАЦІЇ КОДУ

1.1. Дослідження предметної області

Модельно-орієнтоване проектування (MDE) є методологією розробки, яка пропагує використання моделей як основних артефактів інженерії програмного забезпечення. Використовуючи знання предметної області, для застосувань MDE часто розробляються доменно-специфічні мови (DSLs), що дозволяє розробку на вищому рівні абстракції. У середовищі MDE з DSLs моделі, розроблені на DSL, валідуються та трансформуються в програми на загальнодоступній мові програмування (GPL - general-purpose language) через генератор коду. Тестування генераторів коду є складним і зазвичай виконується вручну, що збільшує зусилля та знижує точність тестування.

Модельне тестування (MBT - Model-based Testing) є підходом до автоматизації деяких аспектів тестування. Він реалізується шляхом введення формальних специфікацій програмного забезпечення в інструмент MBT, дозволяючи йому генерувати тести та виконувати кожен слід проти реалізації через адаптер. На основі специфікацій інструмент MBT прогнозує вихід для слідів та перевіряє його відповідність реалізації.

ASOME є сімейством DSLs, що супроводжується генератором коду. Цей генератор коду відображає статичну та динамічну семантику DSLs ASOME. Однак семантика була задокументована та представлена неформально.

Метою цієї роботи є дослідження доречності формальної верифікації та MBT у контексті генератора коду. Для DSL, який генерується в код C++, які є можливості верифікації з MBT? У цьому випадку формальна семантика DSL не задана, тому які кроки потрібно зробити? Нарешті, враховуючи набір формальних специфікацій, чи можна перевірити за допомогою програмного

забезпечення для дослідження моделей, такого як Alloy, чи узгоджені задані специфікації?

Враховуючи використання мови специфікацій Alloy як підходу до формальної верифікації, ця дисертація робить висновок, що семантику DMDSL можна перевести на Alloy, а динамічну семантику можна перевірити. Ця робота є початковою точкою для формалізації специфікацій мови ASOME та перевірки її динамічної семантики. Вона також виводить, що включення формальної верифікації або підходу на основі MBT у розробку на ранніх етапах (тобто з низькою складністю семантики) може дозволити розробникам мов роздумувати над контр-інтуїтивними моделями, які можуть бути пропущені при ручному тестуванні.

1.2. Концепція тестування на основі моделі

Сучасні програмні системи є складними і базуються на великих кодових базах. Такі складні системи нелегко перевірити на правильність. Таким чином, абстрагування щодо деяких аспектів розробки може бути корисним і зменшити людино-години, необхідні для розробки та підтримки систем. У цій мірі методи, засновані на моделях, є відповідним інструментом, що полегшує розробку програмного забезпечення з більш високого рівня абстракції, що дозволяє зменшити складність для розробників.

У модельно-керованій інженерії (MDE) модель є фундаментальною одиницею структурування інформації [14, 19]. Методи, керовані моделлю, засновані на створенні та використанні моделей домену для обміну інформацією. Доменно-орієнтовані мови (DSL) часто використовуються в MDE для визначення моделей домену в більш простому синтаксисі [3]. Це дозволяє інженерам, які мають досвід у домені, висловлювати проекти знайомими мовами. Якщо DSL можна проаналізувати та скомпільувати, то можлива генерація коду та перевірка. DSL часто інтегруються в робочі середовища мови в середовищах розробки (IDE), таких як Eclipse [11].

Зазвичай моделі, розроблені в DSL, перевіряються та перетворюються на мову загального призначення (GPL), наприклад, C++ або Java для їх виконання. Це робиться шляхом розробки генератора коду, який є типовим екзогенним перетворенням моделі [16] - той самий вміст моделі виражається іншою мовою. Таким чином, користувач DSL повинен знати лише семантику DSL, яка є вищою абстракцією, ніж згенерований код. Крім того, моделі також можна створювати за допомогою графічного редактора (в IDE). Загалом, DSL завжди матиме статичну семантику, а також може мати динамічну семантику. Статична семантика включає визначення та призначення моделі в DSL, тоді як динамічна семантика відображає модель DSL на відповідну поведінку виконання («виконання») [23].

Надійність DSL для визначення дійсних моделей залежить від правильності генератора коду. Недоліки в перетворенні моделі можуть призвести до некомпілюваного згенерованого коду; альтернативно ¬, логічні помилки в синтаксично правильно згенерованому коді можуть призвести до непередбачуваних неузгодженостей під час виконання. Як правило, коректність генератора коду перевіряється вручну: або шляхом перегляду перетворення моделі (M2M, M2T [5]); або надаючи вхідні моделі та або перевіряючи згенерований код, або запускаючи тестові випадки на вбудованій версії згенерованого коду. По суті, ручне тестування призводить до нижчої точності та потребує додаткових виправлень пізніше. Можливість покращення тестування, щоб воно було більш надійним, може пізніше дозволити зосередитися на нових функціях оскільки немає помилок, які потрібно виправити.

Тестування на основі моделі (MBT) — це підхід до тестування «чорної скриньки», який підтримує формальну модель програмного забезпечення поряд із реалізацією, щоб забезпечити більшу точність тестування. Термін «специфікації» часто використовується замість формальних моделей. У специфікації визначено очікувану поведінку програмного забезпечення. Вони порівнюються з реалізацією, яка на практиці називається

впровадженням/тестовою системою (IUT/SUT). Існує два методи тестування IUT - онлайн і офлайн. У автономному методі спочатку генеруються всі тести, потім агрегований набір тестів перекладається та виконується на SUT. У підході до онлайн-тестування генерація та виконання тестів є одночасними, але для перекладу та виконання тестів на SUT потрібен адаптер.

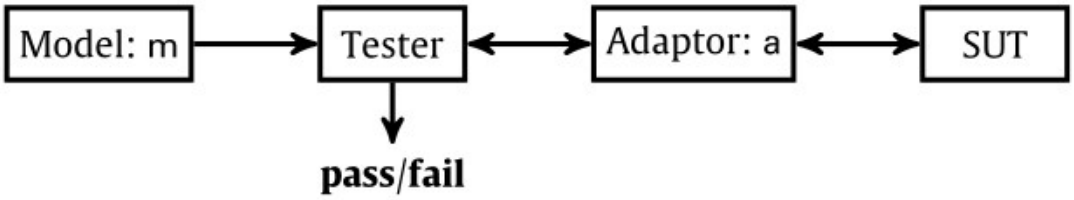

Рис. 1.1. Алгоритм тестування на основі моделі

Налаштування MBT у контексті онлайн-тестування включає три компоненти, як показано на рисунку 1.1. Це формальна модель m, набір тестувальників і адаптер a. Формальна модель m визначає формальні специфікації передбачуваного визначення реалізації. Потім компонент тестувальника аналізує m і генерує для нього тестові випадки як послідовність дій («слідів») у м, починаючи з дійсного початкового стану. Нарешті, необхідно визначити адаптер a, який перетворює тестовий приклад у формат, розпізнаний реалізацією, і виконує його проти IUT. У випадку DSL цей адаптер створює відповідний файл GPL (наприклад, C++). Зауважте, що, як наслідок, MBT у випадку DSL також вимагає, щоб DSL мав динамічну семантику.

У моделі m формальні специфікації повинні бути визначені на основі відповідної парадигми. Існує кілька парадигм для опису формальних специфікацій [25]. Парадигми специфікацій на основі переходів і специфікацій на основі стану зазвичай можна побачити в програмному забезпеченні для перевірки моделі та тестування на основі моделі. Специфікації на основі переходів підходять для реактивних систем [7], які

використовують мічені системи переходів (LTS) [24] або діаграми станів [13] для визначення формальних специфікацій.

По суті, специфікації на основі переходів підходять, коли поведінка IUT базується на вхідних діях і реакціях системи, наприклад, торговий автомат. Як альтернатива, специфікації на основі стану [25] визначають вміст стану як формальну специфікацію, наприклад, у формі наборів або впорядкованих послідовностей. У цьому випадку переходи між різними станами вимагають змін у вмісті станів. Прикладом може бути файлова система з операціями CRUD (Create, Retrieve, Update, Delete) над файлами та папками, де стан визначається набором вмісту файлової системи. Хоча ці парадигми збігаються, акцент робиться або на вмісті системи SUT («властивості системи»), або на реакції SUT («що робить система»). (Для точного порівняння парадигм, заснованих на стані та на основі перехідного періоду, зверніться до [18 , 22]). Для тестування DSL і генератора коду можливо використовувати формальні специфікації на основі стану або переходу - це залежить від семантики DSL і його поведінки під час виконання.

1.3. Дослідження та опис спеціалізованого середовище моделювання програмного забезпечення

Середовище моделювання програмного забезпечення ASML ASOME було розроблено як програмне забезпечення за шаблоном ASML DCA (розділення даних, керування, алгоритмів). Шаблон DCA базується на припущенні, що алгоритми, елементи керування та дані краще перевіряти окремо, а потім об’єднувати. Це базується на дійсності таких інструментів, як ASD і CoCo . Використовуючи ASOME, інженери можуть визначати дані та поведінку в окремих середовищах. ASOME розробляється з 2015 року, і завдяки його практичному використанню реалізація розроблена, але формальні специфікації не визначені.

ASML ASOME (ASML Software Modeling Environment) — це спеціалізоване середовище моделювання програмного забезпечення, розроблене на основі архітектурного шаблону ASML DCA (Design Control Architecture). Його основною метою є підтримка процесу розробки програмного забезпечення для високотехнологічних систем, таких як обладнання для виробництва напівпровідників, що виготовляється компанією ASML.

Наведемо мсновні характеристики середовища ASML ASOME:

- Шаблон ASML DCA (Design Control Architecture). ASML DCA — це архітектурний шаблон, розроблений для структурованого контролю складних систем. Він визначає чітку модульну архітектуру з акцентом на управління складними елементами системи через стандартизовані інтерфейси. Завдяки використанню цього шаблону, ASML ASOME забезпечує модульну побудову програмного забезпечення, що сприяє підвищенню надійності та масштабованості систем.

- Моделювання систем високої складності. ASOME розроблено для створення моделей та симуляцій систем, які є надзвичайно складними, наприклад, обладнання для літографії. Середовище дозволяє створювати детальні моделі апаратного забезпечення та програмних компонентів, що використовуються в сучасних високотехнологічних системах.

- Компонентно-орієнтоване програмування. Одним із ключових аспектів ASOME є підтримка компонентно-орієнтованого підходу до розробки програмного забезпечення. Це означає, що програмні модулі можна легко заміняти або оновлювати окремо, без необхідності значних змін в інших частинах системи.

- Інтеграція з інструментами розробки. ASML ASOME інтегрується з іншими інструментами для розробки програмного забезпечення, зокрема системами контролю версій, тестування та валідації, що робить процес розробки більш ефективним і контрольованим.

- Автоматизація процесів генерації коду. ASOME підтримує автоматизоване перетворення моделей систем у програмний код. Це сприяє прискоренню розробки та мінімізує кількість помилок, пов'язаних із ручним написанням коду.

- Модельно-орієнтоване тестування. Середовище ASOME також дозволяє проводити тестування на основі моделей. Це допомагає забезпечити надійність та коректність роботи програмного забезпечення ще на етапі його розробки, що є критично важливим для систем із високими вимогами до безпеки та надійності.

Переваги використання ASML ASOME наступні :

- Скорочення часу на розробку: Завдяки автоматизації процесів та компонентно-орієнтованій архітектурі.

- Модульність та масштабованість: Можливість легкої адаптації програмного забезпечення до нових вимог без значних переробок.

- Надійність та контроль якості: Підтримка формальних методів і автоматизованого тестування забезпечує високу якість продукту.

ASML ASOME є потужним середовищем, яке сприяє ефективній розробці програмного забезпечення для високотехнологічних та критично важливих систем, таких як напівпровідникові технології.

Фреймворк SAMOS - Statistical Analysis of MOdelS (Статистичний аналіз моделей) є інструментом, розробленим для великомасштабного аналізу моделей за допомогою комбінації методів інформаційного пошуку, обробки природної мови та статистичного аналізу.

Робочий процес аналізу моделей у SAMOS представлений на рисунку 1.2. Процес починається з введення колекції моделей, які відповідають певному метамоделю. Дотепер SAMOS використовувався для аналізу, наприклад, метамоделей Ecore [8] і моделей функцій [9].

Задано колекцію моделей1, SAMOS спочатку застосовує схему вилучення, специфічну для метамоделю, щоб отримати функції цих моделей та зберегти їх у файлах функцій. Функції є атрибутами елементів моделі, які

вважаються актуальними для порівняння таких елементів з метою виявлення клонів. Як показано на рисунку 1.2, SAMOS представляє функції у вигляді N-грам або метрик [5]. Після вилучення файлів функцій наступні кроки є незалежними від типу моделі. Вилучені функції з файлів функцій проходять крок обробки природної мови для токенізації та лематизації значень цих функцій для справедливого порівняння на основі мови. Функції порівнюються для побудови векторної моделі простору (VSM) після призначення схеми зважування атрибутам функцій на основі типу функції, що порівнюється.

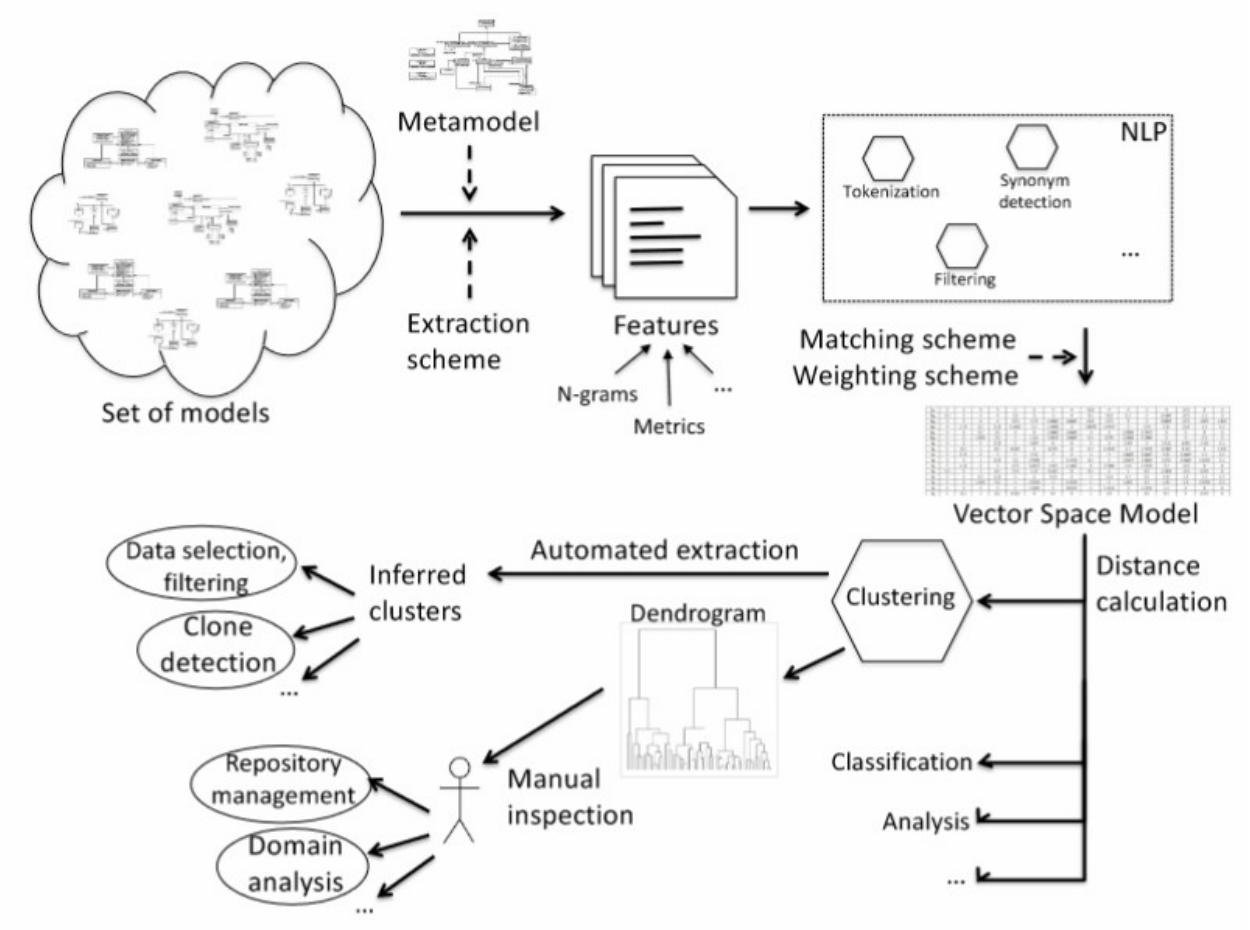

Рис. 1.2. Робочий процес аналізу моделей

«Моделювання інтерфейсу домену DSL» (DMDSL) є частиною сімейства DSL ASOME для обробки аспекту даних шаблону DCA. Існують інші DSL для моделювання керування, алгоритмів і систем, але вони не

розглядаються в цьому проекті. Тут мета полягає в тому, щоб виявити придатність формальної перевірки та MBT для ASOME для підмножини його семантики та перевірити правильність генератора коду.

Наступна діаграма (рисунок 1.3) ілюструє приклад моделі даних, розробленої за допомогою фреймворку ASOME.

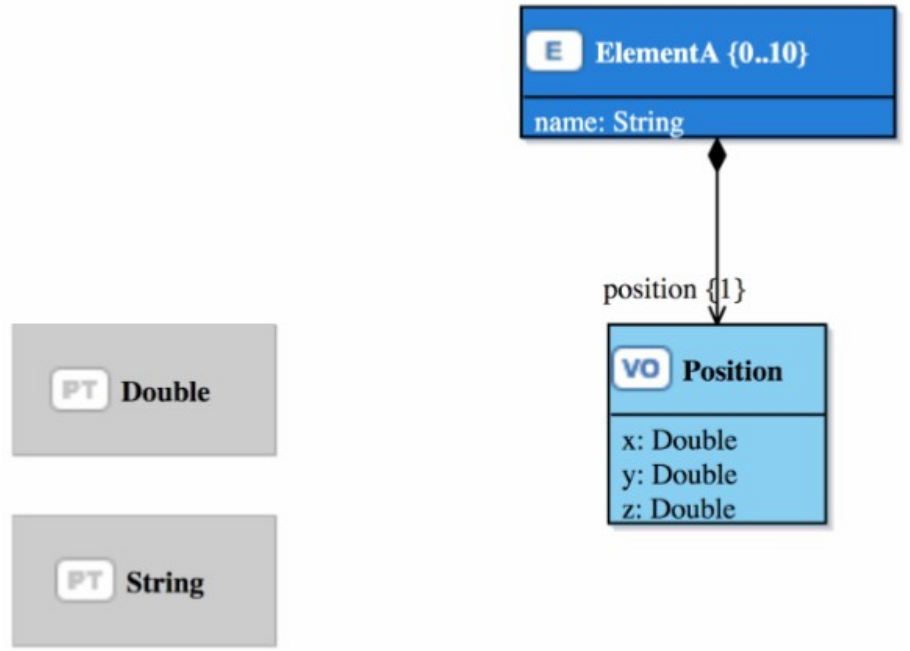

Рис. 1.3. Приклад моделі даних, розробленої за допомогою ASOME

```
Model BasicModel{

DomainInterface iDomain1DM {
        Entity ElementA [0, 10] {
            lifecycle : Constructable Immutable Undeletable Volatile
            attributes :
                name : BasicModel.iDomain1DM.String [1, 1] unordered;
                position : BasicModel.iDomain1DM.Position [1, 1] unordered;
        }
        Type String;
        Type Double;
        ValueObject Position {
            attributes :
                x : BasicModel.iDomain1DM.Double [1, 1] unordered;
                y : BasicModel.iDomain1DM.Double [1, 1] unordered;
                z : BasicModel.iDomain1DM.Double [1, 1] unordered;
        }
    }
}
```

Рис. 1.4. Приклад коду моделі даних

У DMDSL користувач визначає моделі, які представляють інформацію в таких конструкціях, як сутності та ValueObjects , через інтерфейси домену. Відповідно до динамічної семантики, сутності можуть бути створені, і ці екземпляри можуть бути збережені в репозиторії, а вміст репозиторію маніпулюється (витягується, оновлюється, видаляється) під час виконання.

Це операції CRUD+A над сутностями та екземплярами. За аналогією з об'єктно-орієнтованим програмуванням (ООП) сутності подібні до класів, а екземпляри під час виконання подібні до об'єктів класу. Поняття кратності використовується для введення обмежень на кількість екземплярів. Крім того, між деякими конструкціями можна створювати зв'язки : це бінарні зв'язки з доменом і діапазоном, але зв'язки також можуть мати властивості. Статичні обмеження визначаються за допомогою декларативної мови OCL . Ці обмеження використовуються для забезпечення валідності в DMDSL, щоб неузгоджені моделі не можна було скомпілювати в згенерований код.

Зазвичай, коли згадуються ліцензії GPL, такі як C++ або Java, очікується, що час виконання означає «після виконання байт-коду». У контексті DMDSL час виконання відноситься до дій і функцій, записаних у згенерованих файлах C++, які пізніше будуть скомпільовані в байт-код і виконані. Коротко кажучи, динамічна семантика ASOME дозволяє виконувати такі операції, як створення, додавання, отримання, оновлення та видалення примірників у сховищі. Вони називаються операціями CRUD+A . Важливо переконатися, що враховуючи узгоджену поведінку сховища, виконання будь-якої з цих операцій все ще призводить до узгодженої поведінки. Тому необхідне точне визначення цих операцій у формі формальної специфікації. DMDSL детально описано в наступному розділі.

1.4. Постановки проблематики дослідження

Статичну та динамічну семантику DMDSL наведено неофіційно в тексті та на діаграмах. Використання природної мови для обговорення

семантики може спричинити неправильне уявлення між мовними інженерами, що може призвести до прогалин у реалізації. Це дозволяє створювати неузгоджені моделі, які слідують визначеній статичній семантиці DMDSL, але призводять до неочікуваної поведінки під час виконання. Оскільки генератор коду в ASOME тестувався вручну, це свідчить про те, що деякі статичні обмеження моделі можуть бути упущені або що операції в динамічній семантиці можуть потребувати уточнення. Отже, можна визначити суперечливі моделі. (Визначення невідповідності розглядається пізніше.)

Через відсутність офіційних специфікацій DSL, ключовим кроком у цьому проекті є розуміння та формальне визначення семантики. Через часові обмеження та природну складність повну формальну специфікацію ASOME неможливо задокументувати. Застосовується лише підмножина статичної семантики з відповідною динамічною семантикою для узгодженості репозиторію.

Основна мета полягає в тому, щоб розглянути застосовність формальної перевірки для DMDSL, який є функціональним випадком реального промислового використання з 2016 року. Формальна перевірка застосовується шляхом визначення семантики DMDSL у формальному інструменті специфікації Alloy. Визначаючи логічні твердження для узгодженості, можна досліджувати специфікації DMDSL і виявляти моделі, які порушують твердження. Потім потрібно звернути увагу на формальні специфікації або реалізацію, щоб виправити контекст порушення.

Для формальної перевірки генератора коду за допомогою інструменту необхідно вибрати відповідне рішення MBT. У цьому інструменті можна моделювати формальні специфікації та генерувати траси. Вимоги до офіційного інструменту перевірки для тестування специфікації DMDSL задокументовано. У цій роботі Alloy використовується для перевірки специфікації DMDSL. Враховуючи зрілість ASOME та те, як її правильність (і складність) поступово зростала, можна виявити переваги та недоліки

використання програмного забезпечення Alloy у промислових масштабах. З огляду на постановку проблеми та підхід, виділено наступні питання дослідження.

Основні питання дослідження:

1. Як має бути точно визначена семантика DMDSL, враховуючи неформально визначену семантику?

2. Що є придатним інструментом для формальної перевірки семантики DMDSL і що потрібно?

3. Які переваги та недоліки підходу формальної перевірки в реальному промисловому контексті, такому як DMDSL?

Висновки до розділу

У першому розділі роботи було проведено детальний теоретичний огляд передумов моделювання процесів генерації коду. Дослідження предметної області підтвердило, що ефективне моделювання процесів генерації коду потребує систематизованого підходу до управління складністю, особливо в умовах сучасних високотехнологічних середовищ. Виявлено, що традиційні підходи до генерації коду не повною мірою відповідають вимогам поточних технологічних розробок, що створює необхідність в удосконаленні існуючих методів і інструментів.

Концепція тестування на основі моделі розкрита як одна з перспективних стратегій забезпечення якості програмного забезпечення, яка дозволяє зменшити кількість помилок ще на етапі проєктування. Тестування на основі моделі сприяє автоматизації перевірок та підвищує точність і відповідність кінцевого коду специфікаціям.

Аналіз спеціалізованих середовищ моделювання програмного забезпечення показав, що сучасні інструменти, такі як ASML ASOME, надають потужні можливості для структурованого підходу до генерації коду, інтегруючи формальні методи і модельно-орієнтоване тестування. Це

забезпечує більш високу надійність та адаптивність програмних систем. Важливим аспектом є також автоматизація перетворення моделей у програмний код, що значно скорочує час розробки.

Постановка проблематики дослідження охоплює питання підвищення ефективності процесів генерації коду через впровадження сучасних методологій і інструментів. Зокрема, було зазначено, що одним із ключових викликів є необхідність забезпечення якості та масштабованості систем, одночасно знижуючи витрати на розробку та підтримку. Формальні специфікації та моделі стають критичним елементом для досягнення цих цілей.

Таким чином, розділ надає комплексний теоретичний фундамент для подальшого дослідження ефективних методологій моделювання процесів генерації коду, підтверджуючи важливість інтеграції формальних методів у процес розробки програмного забезпечення.

РОЗДІЛ 2. ДОСЛІДЖЕННЯ СТАТИЧНОЇ ТА ДИНАМІЧНОЇ СЕМАНТИКИ ДЛЯ ПРОЦЕСІВ МОДЕЛЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Специфікація середовище моделювання програмного забезпечення ASOME

ASOME розроблено в Eclipse IDE і включає статичну та динамічну семантику (також звану семантикою часу виконання) для складових DSL. Для створення моделей передбачено текстове та графічне зображення. ASOME включає сімейство DSL, розроблених на Altran і ASML. Вони відображають шаблон архітектури алгоритмів керування даними (DCA), прийнятий ASML. Цей текст стосується лише DMDSL. DSL (DMDSL) використовується для визначення моделей даних.

Хоча детальне пояснення семантики DMDSL буде наведено в наступних розділах, тут подано короткий вступ, починаючи з підходу зверху вниз. Першою конструкцією в моделі DMDSL є репозиторій. Це сховище має специфікацію служби та реалізацію служби. Навіть для найпростіших моделей завжди генерується реалізація за замовчуванням. Служба специфікації сховища може визначати принаймні 1 інтерфейс домену. Тоді інтерфейс домену може вказати модель даних (сутності, об’єкти значень, атрибути, примітивні типи, перерахування та константи) та їхні зв’язки (асоціації, спеціалізації та композиції). Відношення є такими: асоціації є односпрямованими від вихідної сутності до цільової сутності та множинність асоціацій як для джерела, так і для цільового; спеціалізація – це однонаправлене відношення для визначення ієрархій та успадкування між об’єктами; Композиція — це відношення, яке виражає вміст цільового ValueObject у вихідному Entity або ValueObject.

Якщо визначена специфікація служби репозиторію дійсна, її код може бути згенерований. У цьому згенерованому коді сутності можна створити та

додати до сховища, яке є механізмом зберігання сутностей. Операції CRUD можна виконувати для маніпулювання репозиторієм, тобто екземплярами в репозиторії. Це аналогічно класам і екземплярам в об'єктно-орієнтованому програмуванні. Після створення екземпляр сутності все ще не є значущим, доки його не буде додано та збережено в сховищі, після чого над ним можна виконувати інші операції. Серед усіх об’єктів даних згенерований код містить ідентифікатори екземплярів сутності. Навпаки, об’єкти-значення не мають ідентифікаторів і не зберігаються в репозиторії, вони містяться лише в сутностях. У згенерованому коді операції CRUD використовуються для екземплярів сутності та їх асоціацій. Множинність сутності обмежує кількість екземплярів сутності, які можна створити. Множинності асоціацій (з такою ж інтерпретацією, як і в UML) вимагають певної кількості екземплярів для участі в асоціації, так що зв’язок асоціації задовольняється. Під час виконання необхідно дотримуватися множинності сутності та асоціації, інакше повертається виняток. Зауважте, що якщо мінімальна кратність для сутності відмінна від нуля, тоді екземпляри цієї сутності повинні бути надані під час створення репозиторіїв, які їх містять.

Статичні обмеження використовуються для забезпечення дійсності моделі, інакше операції під час виконання можуть стати непослідовними або помилковими. Модель дійсна, якщо вона не порушує жодних статичних обмежень. Для забезпечення статичних обмежень використовується мова обмежень об’єктів (OCL) [26].

OCL — це декларативна мова, яка використовується для визначення правил у Meta-Object Facility (MOF) метамоделі, включаючи UML. Eclipse Modeling Framework (EMF) також надає MOF, до якого застосовуються обмеження OCL. Таким чином розробники DMDSL можуть запобігти створенню користувачами недійсних моделей. Правильність обмежень OCL істотно впливає на коректність DMDSL. Обмеження є декларативними і можуть бути такими простими, як забезпечення того, щоб максимальна кратність була більшою за мінімальну кратність, а також складні рекурсивні

виклики, такі як обмеження антициклічності, які повинні перевіряти замкнуті шляхи різної довжини для двійкового (або вищого порядку)) відносини.

Якщо модель задовольняє всі статичні обмеження OCL, для неї може бути згенерований код. Це відомо як дійсна модель щодо семантики DMDSL. Генератор коду компілює модель DMDSL у код C++, після чого вона, як кажуть, перебуває у середовищі виконання, після чого застосовується динамічна семантика (семантика виконання). Можливо, що модель дійсна для DMDSL, але вона може ніколи не бути ініціалізована динамічною семантикою - висновок, що, можливо, потрібні додаткові обмеження, щоб відхилити таку модель під час перевірки.

Наступні розділи пояснюють статичні конструкції в DMDSL та обмеження для забезпечення дійсності моделей.

2.2. Дослідження статичних конструкцій в доменно-специфічній мові моделювання

2.2.1 Специфікація служби сховища

Специфікація служби сховища ('RS' - service specification) є першим елементом, створеним у моделі DMDSL (Domain-Interface Modeling Domain Specific Language). RS визначає всі репозиторії в моделі DMDSL. У моделі може бути кілька RS. Кожен RS має порт, скажімо p, який використовується для підключення до всіх інтерфейсів домену, наданих на цьому порту. Зверніть увагу, що RS є лише специфікацією вмісту. Реалізація сховища (RR - Repository Realization) обробляє функціональні можливості, пов’язані з впровадженням. RR дозволяє реалізувати варіанти інтерфейсу за допомогою однієї реалізації репозиторію та дозволяє еволюцію моделі. Приклад специфікації (RS) наведено нижче. У цьому прикладі порт «p1» визначає доменний інтерфейс iDomainlDM у специфікації сховища sServicelRS і має неявну реалізацію сховища за замовчуванням, яка не відображається на схемі.

Рис. 2.1. Специфікація служби сховища (RS)

2.2.2. Інтерфейси домену

Служба сховища може надавати або вимагати доменних інтерфейсів через порти. Інтерфейс домену (DI) у DMDSL – це місце, де визначаються основні статичні конструкції. У DI можна визначати сутності, асоціації, об’єкти значень тощо. У моделі може бути кілька DI, і вони можуть надаватися з одного порту. Метою інтерфейсу домену є уможливлення еволюції моделі та розділення клієнтів: сутність з деякими властивостями в інтерфейсі домену може існувати (з тим самим або іншим ім’ям) як сутність в інтерфейсі іншого домену з іншими властивостями. Це включає в себе реалізацію репозиторію, яка не входить до сфери цього тексту.

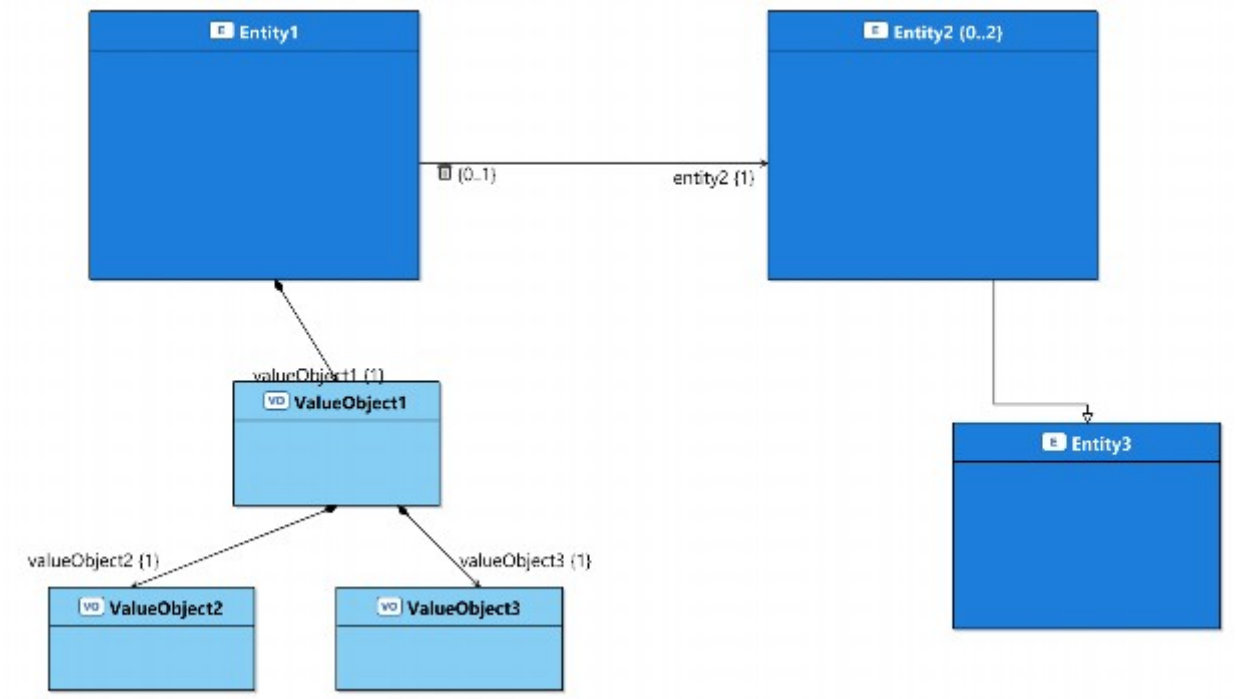

Рис. 2.2. Приклад: вміст в інтерфейсі домену

У цьому тексті розглядається контекст інтерфейсу єдиного домену, що означає, що кожна сутність є окремою та має одне представлення . Інтерфейс домену може мати ряд визначених у ньому конструкцій. Графічний редактор забезпечує відображення всіх властивостей на екрані з текстом і значками, як показано на рисунку 2.2. Зверніть увагу, що різні стрілки вказують, який тип зв'язку визначено. Entity та Entity2 мають зв'язок асоціації. Entity2 і Entity3 мають відношення спеціалізації (Entity3 є батьківською сутністю Entity2, тобто Entity2 успадковує Entity2 за аналогією з ООП). Об'єкти значення містяться в Entityl через відношення композиції.

2.2.3 Опис сутності DMDSL

Сутність у DMDSL використовується для зберігання даних через атрибути, константи та ValueObjects. За аналогією з UML це схоже на клас. Можливо, можна створити екземпляр сутності під час виконання (залежно від її властивостей). Кожен екземпляр сутності ідентифікується унікальним ідентифікатором (два екземпляри сутності можуть бути створені з однаковими властивостями та вмістом, але вони відрізняються). Сутність має такі властивості:

1. Конструктивність. Приймає значення з {Constructable, Unconstructable}. Сутність із властивістю Constructable може бути створена з інтерфейсу домену, у якому вона визначена. Однак, якщо властивість Constructability для сутності є Unconstructable, тоді сутність не може бути створена через інтерфейс , хоча це залежить від реалізації репозиторію він може бути створений з іншого інтерфейсу.

2. Змінність. Приймає значення з {Editable, Undeditable, Immutable}. Якщо Entity має Mutability = Editable в інтерфейсі, тоді його екземпляри можуть бути оновлені під час виконання користувачем, який отримує доступ до екземпляра через інтерфейс. Якщо Mutability = Undeditable в інтерфейсі, екземпляри сутності не можуть бути оновлені користувачем, який отримує доступ до нього з цього інтерфейсу (однак інші інтерфейси, які надають

сутності властивість як Editable, все одно можуть використовуватися для зміни екземплярів). Якщо Mutability = Immutable , екземпляри сутності не можуть бути оновлені ніким після того, як вони зберігаються в репозиторії. Статичні обмеження щодо реалізації репозиторію забезпечують те, що одна й та сама сутність не може бути одночасно визначена як редагована та незмінна.

3. Видалення. Приймає значення з {Deleteable, Undeletable, Undestructable}. Якщо Entity має властивість Deleteability = Deleteable, то його екземпляри можна видалити зі сховища через інтерфейс. Якщо властивість Deleteability = Undeleteable в інтерфейсі, то його екземпляри не можна видалити через цей інтерфейс (хоча користувачі іншого ¬інтерфейсу, де властивість Deleteable, можуть видалити екземпляр). Однак, якщо Deleteability = Undestructable , тоді екземпляри сутності ніколи не можна буде видалити після збереження в репозиторії.

Хоча властивості сутності дозволяють операції над ними, сутність також може мати визначену множинність сутності . Множинність сутності визначає нижню межу (LB/мінімум) і верхню межу (UB/максимум). Множинність обмежує, скільки екземплярів сутності може існувати в певний момент часу, і необхідно перевірити, що динамічна семантика відповідає цьому.

Нижче наведено приклад визначення сутності у графічній формі.

Рис. 2.3. Сутність з атрибутами та властивостями

Властивості сутності (та інших конструкцій DMDSL) можна побачити в IDE:

Property	Value
Entity Entity1	
Constructability	CONSTRUCTABLE
Deletability	UNDESTRUCTABLE
Fqn	ExampleModel.iDomain1DM.Entity1
Kind	DATA
Mutability	IMMUTABLE
Name	Entity1
Persistent	false
Representation	ExampleModel.iDomain1DM.Entity1
Specializes	

Рис. 2.4. Властивості конструкції Entity

Після створення екземпляри сутності можуть зберігатися в сховищі сутностей. Для кожного типу сутності в моделі створюється репозиторій. Загалом термін репозиторій використовується для позначення комбінації всіх сховищ усіх об’єктів. Примірники сутності все ще можуть існувати поза їхніми сховищами, тому множинність вважається такою: для створення примірника дотримується максимальна множинність (незалежно від сховища). Однак мінімальна кратність завжди перевіряється для екземплярів у сховищі. Для екземплярів у сховищах завжди враховується множинність асоціацій.

Передбачувана семантика полягає в тому, що спочатку сховища порожні. Однак у разі мінімальної кратності, відмінної від нуля, репозиторії мають бути заповнені за допомогою -делегатів конструктора. Зверніть увагу на кратність (2..4) Entityl на рисунку 2.3. У таких випадках перед виконанням очікується, що екземпляри цієї сутності будуть заповнені в репозиторії, щоб сутність і множинність асоціацій були дійсними до виконання. Делегат конструктора дозволяє створювати екземпляри та зберігати їх у сховищах, і хоча це все ще робить користувач через програмний інтерфейс у згенерованому коді, вважається, що це перед виконанням. Коли репозиторій ініціалізовано, можна очікувати, що делегати конструктора надали

екземпляри сутності у своїх відповідних репозиторіях, щоб усі сутності та множинності асоціацій були задоволені.

Зауважимо, що делегати конструктора по суті є API C++, як і інший згенерований код. Ці делегати генеруються лише для сутностей, які мають ненульову мінімальну кратність, але в делегатах користувач може створювати екземпляри будь-якої сутності, незалежно від її мінімальної кратності.

2.2.4. Об'єкти Value

На відміну від сутностей, об’єкти Value, по суті, є кортежами атрибутів без ідентифікатора. Відповідно , вони також не зберігаються в репозиторії. Однак об’єкти ValueObjects можуть створюватися та міститися іншими конструкціями (тобто Entities або іншими ValueObjects), і вони успадковують властивість змінності від сутності, що їх містить (відношення Composition, в 2.2.5). Два ValueObjects можуть бути створені з однаковими значеннями, але тоді вони не відрізняються (на відміну від сутностей).

Приклад моделі з ValueObject наведено нижче. У цьому налаштуванні ValueObject2 і ValueObject3 містять деякі цілі представлення компонентів вектора швидкості, а потім ValueObjectl може бути деякою алгоритмічною оцінкою його відповідних атрибутів, наприклад netVelocity є коренем квадратним від суми verticalVelocity^2 + horizontalVelocity^2, отримані від ValueObject2 і ValueObject3.

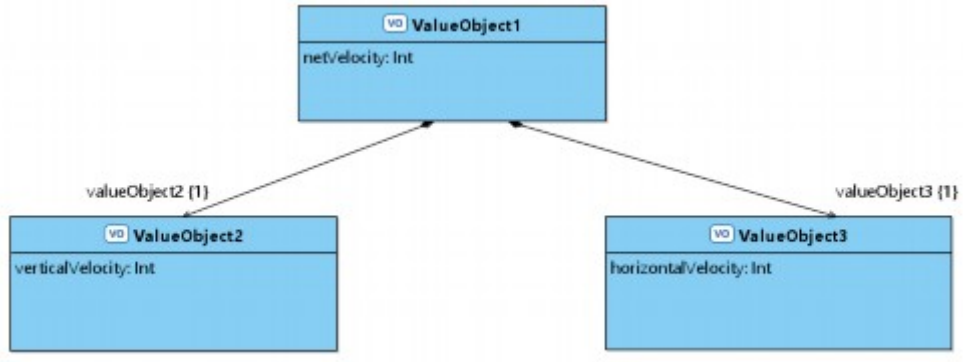

Рис. 2.5. Конструкція об’єктів Value

2.2.5 Типи та характеристики відношень

У DMDSL є три відношення і всі вони є бінарними. Це асоціація, спеціалізація та композиція. Відношення, важливе для цього тексту, — асоціація, оскільки складність зв'язків, враховуючи обмежений час для виконання завдання, ускладнює включення всіх цих зв'язків. Відносини описані нижче.

Асоціація — це бінарне відношення від сутності до сутності. Він односпрямований (джерело-ціль). Семантика подібна до відношення асоціації UML, за винятком того, що сутність не може бути пов'язана сама з собою (нерефлексивна). Асоціація також має властивості: множинність і каскадне видалення. У згенерованому коді асоціації адресуються з екземпляра вихідної сутності.

• Під час виконання асоціація створюється/заповнюється, коли створюється посилання з екземпляра (вихідного типу) на екземпляр цільового типу. Ці зв'язки між екземплярами сприяють множинності джерела та цільової асоціації. Множинність асоціації визначається як на вихідному, так і на цільовому кінцях і вказує на кількість екземплярів, які можуть брати участь у асоціації, пов'язані з даним екземпляром.

• Властивість Cascade Delete визначено з метою явного моделювання життєвого циклу. Каскадне видалення теоретично визначено для вихідного та цільового кінців асоціації, однак розглянута версія ASOME розглядає лише вихідне каскадне видалення (SCD - source cascade delete). Властивість SCD вказує на те, що, враховуючи асоціацію, яка з'єднує вихідний екземпляр із цільовим екземпляром, якщо цільовий екземпляр видаляється, тоді чи слід також видаляти вихідний екземпляр. Зв'язок між джерелом і метою видаляється незалежно від властивості SCD. Але якщо властивість SCD має значення true, вихідний екземпляр також видаляється. Формальна специфікація в цьому тексті також розглядає цільове каскадне видалення (TCD) — семантику для обробки видалення цільового екземпляра, якщо вихідний екземпляр у зв'язку (з увімкненим TCD) видаляється.

Приклад асоціації наведено нижче. Дотримуватись цільової множинності (1..3) — це означає, що екземпляр типу Entityl не може існувати в сховищі без принаймні w1 екземпляра типу Entity2 у репозиторії, оскільки в іншому випадку асоціація порушується. Крім того, піктограма кошика на вихідному кінці асоціації вказує на те, що властивість SCD для асоціації ввімкнено. Отже, якщо екземпляр Y Entity2 буде видалено, тоді всі посилання на associationExample , де Y був цільовим, також видалятимуть вихідний екземпляр.

Рис. 2.6. Відношення асоціації: множинність та каскадування

На діаграмі зображена асоціація між двома класами: Entity1 та Entity2. Асоціація - це зв'язок між двома класами, який показує, як об'єкти цих класів пов'язані між собою.

Ключові елементи діаграми:

- Класи: Entity1 та Entity2 - це назви класів. Класи представляють концептуальні сутності в системі.

- Асоціація: Лінія між класами Entity1 та Entity2 з написом "associationExample" представляє асоціацію між цими класами. Це означає, що між об'єктами цих класів існує певний зв'язок.

- Множинність: Числа в фігурних дужках біля кінців лінії асоціації вказують на множинність зв'язку.

(0..2) біля Entity1: Це означає, що один об'єкт класу Entity1 може бути пов'язаний з нулем, одним або двома об'єктами класу Entity2. Тобто, один об'єкт Entity1 може мати від 0 до 2 пов'язаних з ним об'єктів Entity2.

(1..3) біля Entity2: Це означає, що один об'єкт класу Entity2 може бути пов'язаний з мінімум одним і максимум трьома об'єктами класу Entity1. Іншими словами, один об'єкт Entity2 повинен бути пов'язаний хоча б з одним об'єктом Entity1, але може бути пов'язаний максимум з трьома.

Символ сміттєвого кошика біля асоціації з боку класу Entity1 вказує на те, що якщо видалити об'єкт класу Entity1, то пов'язані з ним об'єкти класу Entity2 не будуть автоматично видалені. Тобто, видалення об'єкта Entity1 не впливає на існування пов'язаних з ним об'єктів Entity2.

Навігація по асоціації з вихідного екземпляра повертає невпорядковану колекцію екземплярів, яка допускає дублікати, і це відомо як семантика сумки. У реалізації також можливі впорядковані асоціації, що вказують на послідовність екземплярів. В обох випадках допускаються дублікати, тобто той самий цільовий екземпляр може зустрічатися кілька разів як ціль асоціації, і асоціація вважається дійсною. Якщо навігація асоціації повертає набір екземплярів , то дублікати не будуть дозволені. Приклад наведено нижче на рисунку 2.7.

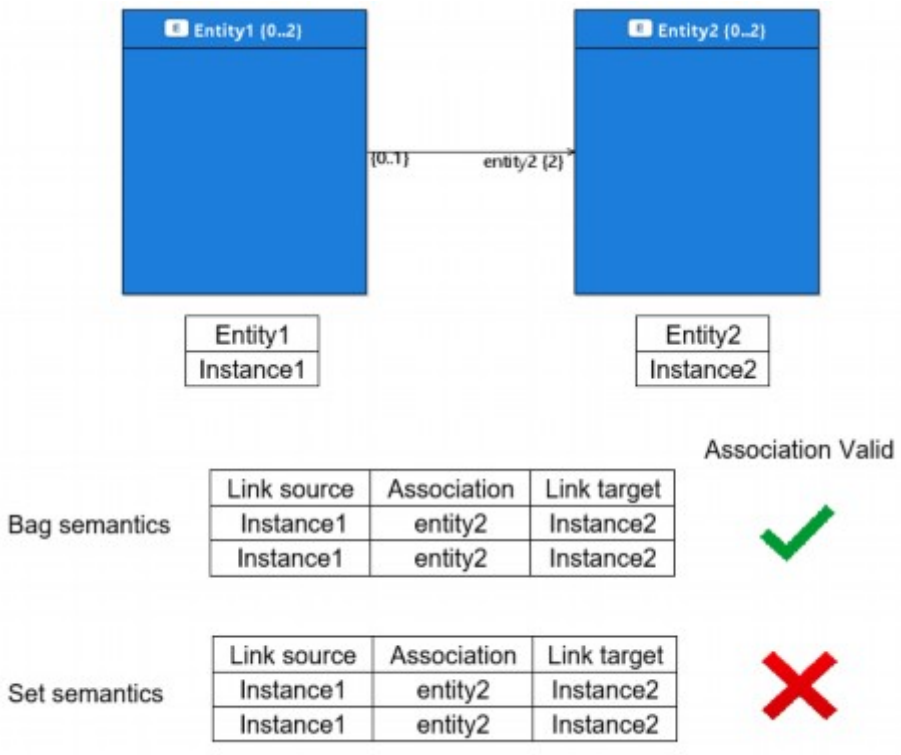

Рис. 2.7. Семантика багатозначних множин проти семантики множин

На рисунку 2.7 представлено порівняння семантики сумки та семантики набору для вірності асоціації. Тут мінімальна цільова кратність асоціації дорівнює 2. У семантиці Bag і Sequence (як передбачає реалізація) можна створити Instance2 і посилатися на нього двічі, вказуючи з Instancel . Це задовольняє мінімальну цільову кратність для Instancel. Як альтернатива, у встановленій семантиці це було б неможливо, і знадобився б інший екземпляр Entity2.

Відношення спеціалізації визначається для введення класифікаційних ієрархій серед сутностей. Це бінарне відношення, де домен і діапазон завжди є сутністю, яка має бути в одному інтерфейсі. Тепер дочірня сутність не може мати більше однієї батьківської сутності. Батьківська сутність називається спеціалізованою сутністю, а дочірня сутність — спеціалізованою сутністю. Будь-яка спеціалізована сутність є абстрактною у реалізації та не може бути створена. Усі властивості, атрибути, асоціації успадковуються спеціалізованою сутністю. Відношення спеціалізації не має такі властивості, як множинність, але це вводить обмеження на множинність сутностей-учасників: мінімальна та максимальна множинність батьківської сутності має бути сумою всіх її спеціалізованих сутностей.

Приклад відношення спеціалізації наведено нижче. Звернемо увагу на асоціацію assocO, успадковану в Entity1_Child. Верхня межа Entityl , 5, є сумою всіх дочірніх сутностей = 5.

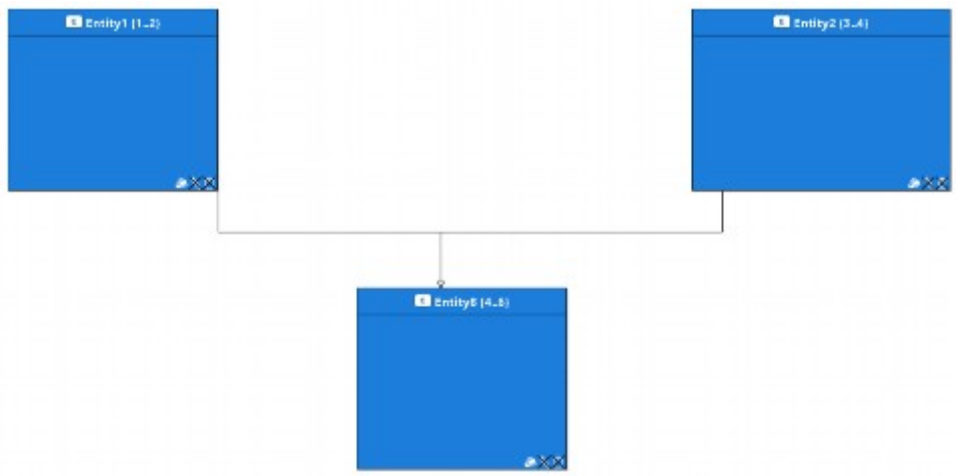

Рис. 2.8. Відношення спеціалізації

Відношення композиції визначає відношення вмісту ValueObject до (Entity або ValueObject), тобто метою композиції є ValueObject. Композиція від Entity до VO по суті представляє VO як атрибут містить Entity. Так само ValueObject може містити інший об’єкт. Таким чином, метою композиції завжди є ValueObject. Приклад композиції наведено вище на рисунку 2.5.

2.2.6. Предметна орієнтованість сховища

У DMDSL репозиторії можуть бути визначені у двох орієнтаціях: еталонна орієнтація та орієнтація клонів. При запиті для сутності орієнтоване на посилання репозиторій надає посилання на вміст. Оновлення вмісту є неявним і миттєвим, що підвищує ефективність, але може призвести до перешкод між клієнтами, крім того, відкат ускладнюється. У репозиторії, орієнтованому на клонування, створюється та повертається клон екземпляра, це означає, що оновлення потрібно зберігати явно. Це дозволяє підготувати дані перед фіксацією, відкатом і ізоляцією клієнтів. Орієнтація встановлюється для кожного типу сутності в реалізації.

Враховуючи доступний час і складність семантики, передбачається, що всі репозиторії надають орієнтовані на посилання сутності. Це спрощує пошук сутності, оскільки не потрібно звертатися до клонів.

2.2.7. Методика розділення інтерфейсу та реалізації

Оскільки в цій роботі розглядається лише реалізація за замовчуванням/створена, цей розділ не має відношення до формальної специфікації.

Служба специфікації сховища може надавати більше одного інтерфейсу домену. Наприклад, розглянемо випадок, коли один клієнт може створювати або оновлювати сутності з одного інтерфейсу, а інший клієнт може лише читати сутності з інших інтерфейсів. Через реалізацію сховища (RR) інтерфейс домену може надавати клієнту лише частину реалізації. RR визначається для даної специфікації (RS) і забезпечує реалізацію для всіх

елементів у всіх наданих інтерфейсах. Реалізація містить усі елементи доменного інтерфейсу. RR реалізує ValueObjects, Entities, відносини. Однак елемент реалізації може реалізовувати більше ніж один елемент інтерфейсу, як показано на рисунку 2.9.

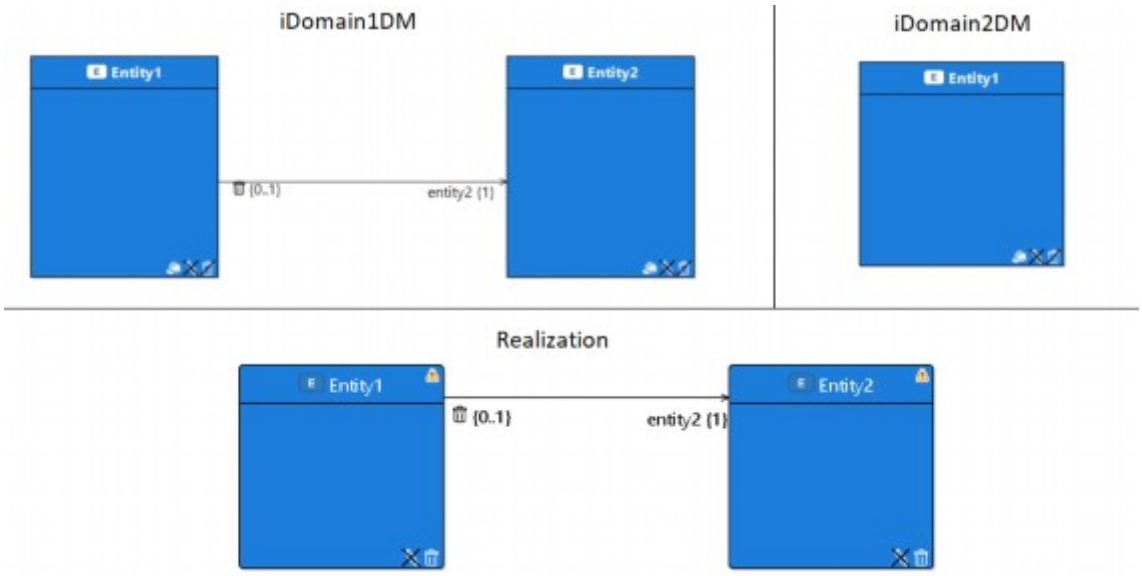

Рис. 2.9. Властивості реалізації сховища

В iDomainlDM є дві сутності та асоціація. В iDomain2DM є одна сутність, яка також є джерелом асоціації в iDomainlDM. Реалізація визначає властивості для всіх сутностей і асоціацій, виявлених у специфікації. Тепер Entity в iDomain2DM може бути дозволено видалятися, і модель все ще буде дійсною. Проте, користувач із доступом до iDomainlDM не може видаляти екземпляри Entity , оскільки для них Entity неможливо видалити.

У Реалізації, наведеній вище, Entityl реалізує iDomainlDM.Entityl та iDomain2DM.Entity1.

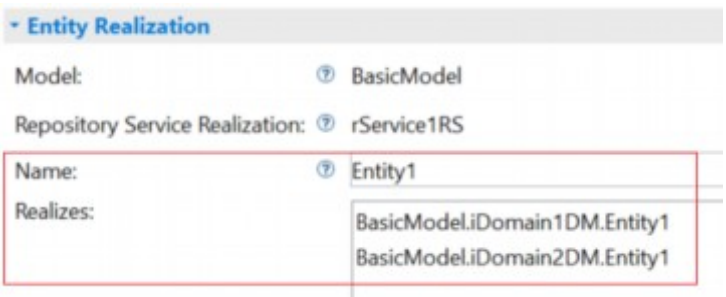

Рис. 2.10. Властивості реалізації

Цей фрагмент інтерфейсу дозволяє визначити конкретну сутність в рамках певної моделі. Сутність "Entity1" є реалізацією двох більш абстрактних сутностей і використовує сервіс "rService1RS" для зберігання своїх даних.

У семантиці RR для Entity немає властивості Constructability. Множинність повинна мати однакові значення в інтерфейсі та реалізації. Властивість Mutability сутності реалізації може бути тільки Editable або Immutable, а властивість Deletea bility сутності реалізації може бути лише Deleteable або Undestructable. Таким чином, властивість Mutability як Editable або Uneditable в інтерфейсі відображається на Editable у реалізації; і Immutable в інтерфейсі відображається на Immutable в реалізації. Подібним чином властивість Deleteability як Deleteable або Undeleteable в інтерфейсі відображається на Deleteable в реалізації, а Undestructable в інтерфейсі відображається на Undestructable для реалізації. Відношення асоціації також має бути реалізовано - якщо асоціація має Cascade Delete, увімкнену в інтерфейсі, вона повинна мати каскадне видалення, увімкнену в реалізації. Вищезгадана семантика забезпечується статичними обмеженнями.

Специфікація сховища лише з одним наданим інтерфейсом завжди може мати реалізацію за замовчуванням, автоматично згенеровану для генерації коду. Реалізація за замовчуванням забезпечує відображення 1-1 між інтерфейсом і реалізацією. Для більш ніж одного наданого інтерфейсу можлива реалізація за замовчуванням, але її, можливо, доведеться змінити.

2.2.8. Область застосування конструкцій

У цій роботі сфера застосування обмежена інтерфейсом домену, включаючи асоціації сутностей і як щодо всіх їхніх властивостей і множинності. Відношення спеціалізації та реалізації сховища не розглядаються. Проста дійсна модель DMDSL, заснована на включеній семантиці, надається в графічній і текстовій формі. Ця модель буде використана як приклад роботи.

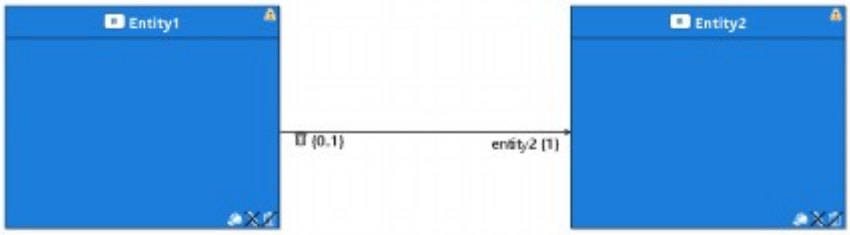

Рис. 2.11. Приклад моделі функцій DMDSL

```
Model BasicModel {
    RepositoryService sService1RS {
        Provided DataPort p1 {
            interfaces : iDomain1DM
        }
    }
    DomainInterface iDomain1DM {
        Entity Entity1 [0, inf] {
            lifecycle : Constructable Immutable Undeletable
            associations :
                [0, 1] entity2 : Entity2 [1, 1] unordered {
                    lifecycle :
                        on source delete : target stays
                        on target delete : source dies
                };
        }
        Entity Entity2 [0, inf] {
            lifecycle : Constructable Immutable Undeletable
        }
    }
}
```

2.3. Дослідження динамічної семантики для процесів генерації коду

Код для валідної моделі ASOME може бути згенерований за допомогою Моделі реалізації - колекції правил. Ми будемо розглядати лише стандартні правила (рецепти) реалізації. На зображенні нижче показано трансформацію моделі генератора коду.

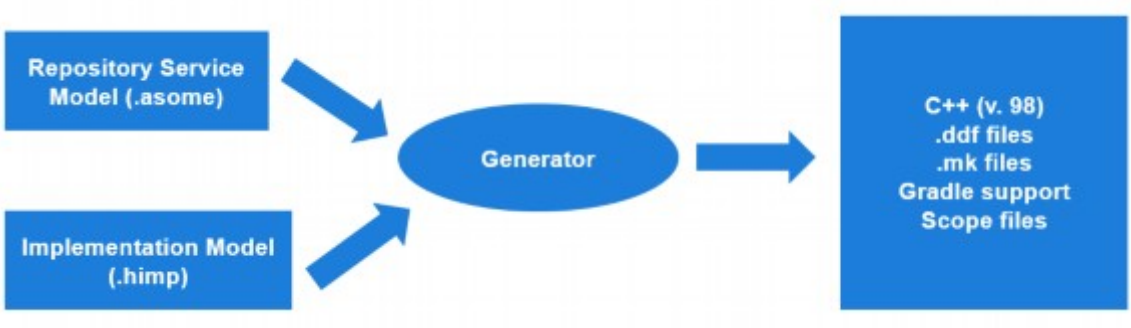

Рис. 2.12. Трансформація генерації коду

Динамічна семантика (час виконання) застосовна до згенерованого коду, визначаючи логіку, яка можлива в коді за допомогою викликів функцій C++. У цій семантиці сутності можна створювати, додавати, отримувати, оновлювати та видаляти, тобто операції CRUD+A. Нижче наведено важливі обмеження щодо динамічної семантики:

• Сутності можна додавати до репозиторію лише по одній. Немає підтримки семантики, подібної до транзакцій (де кілька об’єктів додаються до репозиторію одночасно)

• У будь-який момент може бути виконана будь-яка операція CRUD+A. Очікується, що операції CRUD+A мають поважати узгодженість сховища. Коротко кажучи, визначення узгодженості сховища – це задоволена множина об’єктів і асоціацій.

• Щоб зберегти множинність асоціацій, пам’ятаючи, що може бути лише одна сутність додається до репозиторію за один раз, мінімальна кратність джерела асоціації примусово дорівнює 0. (Хоча мінімальна цільова кратність може бути відмінною від нуля).

Таким чином, ціль завжди можна спочатку додати до сховища, потім джерело, і узгодженість зберігається.

2.3.1. Створення екземпляра сутності

Щоб створити сутність, потрібне посилання на її заводський клас. В межах інтерфейсу домену це можливо, лише якщо тип Entity є Constructable. Для будь-яких вихідних асоціацій, у яких бере участь цей екземпляр, також потрібні цілі асоціації, а виконання множинності цієї асоціації є попередньою умовою. Кількість екземплярів цієї сутності не повинна перевищувати її максимальну кратність, інакше виникає помилка виняткової ситуації під час виконання.

Розглянемо базову модель, представлену на рисунку 2.11. З інтерфейсу iDomain1DM для створення екземпляра Entity1 потрібна наступна логіка. Оскільки Entity1 є джерелом асоціації, його екземпляр не може бути

створений, якщо не існує екземпляра Entity2. Зауважте, що це простий приклад, який використовує лише репозиторій тільки для читання.

Лістинг 2.1. Семантика виконання DMDSL

```
SB = ServiceBundle();          // an sb is always required

E2Factory = SB -> getEntity2Factory();      // get the factory
Instance_E2 = E2Factory -> create();        // create an instance
E2_Repository = SB -> getForEntity2();              // get repository access
E2_Repository -> add(Instance_E2 -> getID());   // Instance_E2 is added now


E1Factory = SB -> getEntity1Factory();              // gets the factory
Instance_E1 = E1Factory -> create(Instance_E2 -> getID());  // 1 instance of
    entity2 required
E1_Repository = SB -> getForEntity1(Instance_E1);    // get repository access
    for Entity1
E1_Repository -> add(Instance_E1 -> getID());     // store instance in
    repository
```

Доповнення до репозиторію показано в лістингу 2.1 з E2_Repository. Коли екземпляр додається до репозиторію, його асоціації мають бути перевірені на (1) валідність множинності джерела (2) валідність множинності цільових даних і (3) усі цільові асоціації цього екземпляра повинні бути у своєму сховищі. Порушення будь-якої з цих умов викликає помилку виконання.

2.3.2. Оновлення та видалення екземпляра із сховища

Екземпляр сутності можна завжди оновлювати через будь-який інтерфейс, якщо він не зберігається в репозиторії. Якщо він зберігається в репозиторії, його можна оновити, якщо він має змінність = Editable. Існує два можливі оновлення екземпляра сутності – оновлення атрибутів і оновлення асоціацій. Оновлення атрибутів аналогічно використанню методів get() і set() в ООП для оновлення об’єктів класу. Це не входить до сфери тексту. Оновлення зв’язків є важливим: за допомогою методу оновлення можна встановити всі цільові екземпляри екземпляра для певного зв’язку. По суті, для даного прикладу лише одна асоціація може оновлюватися під час операції оновлення, і надається нова колекція цілей для асоціації,

перезаписуючи старіші цілі асоціації. Під час операції оновлення важлива перевірка множинності джерела асоціації та цілі.

Екземпляр може бути видалений, якщо він знаходиться в сховищі, а тип сутності має властивість Deleteable в інтерфейсі. Екземпляр усе ще можна видалити, якщо властивість має статус Undeleteable в інтерфейсі, якщо воно має каскадне видалення, яке запускається поза інтерфейсом. Однак, якщо сутність має властивість Незнищуваний, її ніколи не можна буде видалити зі сховища. Під час видалення екземпляра спостерігається наступне:

• Сам екземпляр буде видалено. Передбачувана семантика полягає в тому, що екземпляр буде видалено зі сховища. Реалізація відрізняється від специфікацій щодо цього - у реалізації екземпляр видаляється зі сховища, але все ще існує в пам’яті. Тому всі асоціації екземпляра зберігаються, але вони не враховуються під час перевірки асоціації. У офіційній специфікації видалений екземпляр і його посилання припиняють існування.

• Якщо екземпляр є цільовим посиланням на асоціацію, то вихідний екземпляр буде видалено, якщо для асоціації ввімкнено властивість видалення вихідного каскаду (SCD). В іншому випадку вихідний екземпляр буде оновлено: його посилання на цільовий екземпляр буде видалено.

• Оскільки кратність джерела асоціації має бути мінімум 0, видалення ніколи не порушує кратність джерела. Однак це може порушити цільову множинність, якщо статичні обмеження та функцію видалення не вказано правильно. Таким чином, семантика каскадного видалення є важливою.

Каскадне видалення екземплярів сутності після асоціацій ускладнює операцію видалення. Вважайте, що вихідне каскадне видалення (SCD) увімкнено, тоді необхідно перевірити, що видалення екземпляра цільової сутності не порушує множинність вихідної сутності.

Даний процес забезпечується застосуванням статичних обмежень, як у розділі 2.4. Якщо як SCD, так і TCD увімкнуто для асоціації, необхідно

гарантувати, що навігація через асоціацію не викликає циклу та завжди завершується.

2.4. Представлення статичних обмежень для підвищення якості моделі

Статичні обмеження - це правила, які перевіряються під час створення або редагування моделі. Вони визначають, які елементи моделі можуть бути пов'язані між собою, які атрибути можуть мати певні значення тощо. Ці обмеження допомагають забезпечити коректність і узгодженість моделі.

Ці обмеження визначено, щоб виключити моделі, які можуть спричинити невідповідності під час виконання. У DMDSL існує багато статичних обмежень, заснованих на семантиці реалізацій і кількох інтерфейсах. Розглянута підмножина семантики DMDSL знаходиться в межах одного інтерфейсу. Для цієї підмножини відповідні статичні обмеження перераховані нижче неформальною мовою.

• Кратності: мінімальна кратність має бути невід’ємною, а максимальна – додатною. Максимум має бути більшим або дорівнювати мінімуму.

• Сутності: у кратності сутностей, якщо мінімум дорівнює максимуму, сутність має бути незнищуваною.

• Асоціації:

- Множність джерела асоціації завжди повинна мати мінімум 0.

Оскільки одночасно до репозиторію можна додати лише один екземпляр, це обмеження є необхідним ¬для підтримки узгодженості сховища. За допомогою цього обмеження цільовий екземпляр може бути доданий до репозиторію першим, тоді як нульові вихідні екземпляри вказують на нього. Однак до цілі таке обмеження не застосовується. Таким чином, для джерела в позиції в сховищі може знадобитися > 0 цільових екземплярів у сховищі. Згідно з визначенням асоціації та враховуючи, що лише екземпляр додається у час, вихідна або цільова мінімальна кратність

завжди має бути 0, інакше узгодженість асоціацій порушується для ненульових мінімальних кратностей). Реалізація виконує це на джерелі. На рисунку 2.13 показана недійсна модель, оскільки вихідний мінімум встановлено відмінним від нуля.

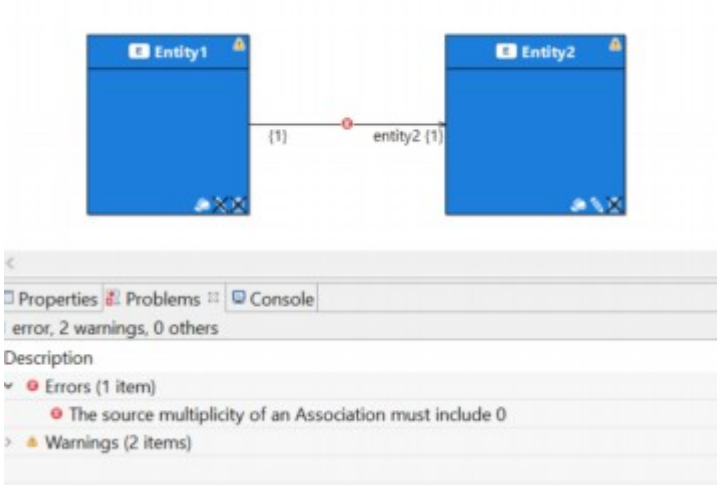

Рис. 2.13. Недійсна модель: мінімальна множинність джерела асоціації має бути 0

- Якщо цільову сутність асоціації можна видалити, то слід увімкнути SCD або зробити вихідну сутність доступною для редагування. Якщо SCD було ввімкнено, то вихідний Entity також повинен мати властивість Deleteable. Крім того, кратність асоціації джерела завжди має мінімум 0, як визначено попереднім обмеженням.

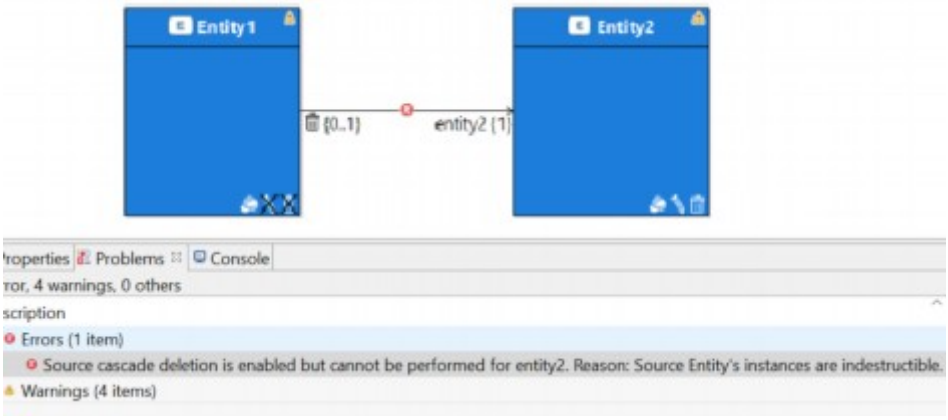

Рис. 2.14. Недійсна модель: SCD увімкнено, але вихідна сутність є неруйнівною

- Якщо вихідну сутність асоціації можна видалити, тоді можна ввімкнути TCD, що примусово видаляє цільову сутність. Крім того, якщо TCD вимкнено, немає проблем зі статичними обмеженнями. Оскільки вихідний екземпляр видалено, асоціація безумовно задовольняється: вихідний мінімум дорівнює 0 і ніколи не може бути порушений, а цільовий мінімум може бути відмінним від нуля, але оскільки вихідний екземпляр перестає існувати, цільова множинність не має передумови.

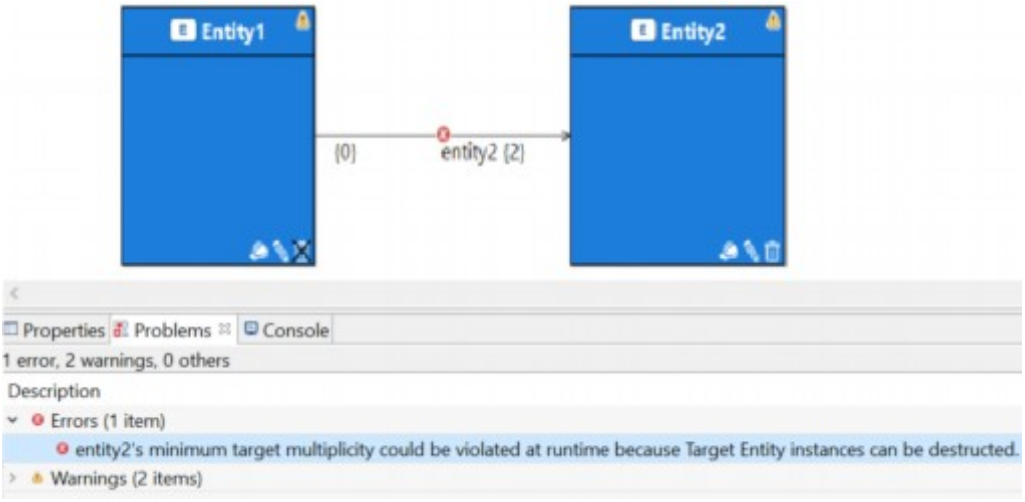

Рис. 2.15. Недійсна модель: екземпляри цілі можна видалити, але кратність цілі = 2, очікується 0

Висновки до розділу

У другому розділі проведено детальне дослідження статичної та динамічної семантики, що застосовується в процесах моделювання програмного забезпечення, зокрема у середовищі ASOME. На основі аналізу специфікацій та конструкцій доменно-специфічної мови моделювання (DMDSL) зроблено такі висновки:

Статична семантика в ASOME забезпечує чітку структуру для опису моделей програмного забезпечення. Це включає такі аспекти, як специфікація служби сховища, визначення інтерфейсів домену, опис сутностей, об'єктів Value, а також типи та характеристики відношень. Такий

підхід дозволяє створювати стабільні моделі, які точно відображають структуру та логіку системи.

Розділення інтерфейсів та реалізацій є ключовою методикою, що підвищує гнучкість та масштабованість системи. Це сприяє зниженню залежностей між компонентами та полегшує подальше внесення змін або розширення функціоналу.

Предметна орієнтованість сховища в моделюванні в ASOME дозволяє відокремити дані на основі їхньої логічної сутності, що підвищує ефективність управління даними. Ця підхід допомагає забезпечити відповідність моделі предметній області.

Динамічна семантика моделі, зокрема можливість виконання операцій над сутностями (створення, оновлення, видалення), забезпечує гнучкість у генерації коду та його управлінні. Це дозволяє здійснювати динамічні операції над даними, зберігаючи відповідність логіці системи. Статичні обмеження відіграють важливу роль у підвищенні якості моделі, запобігаючи помилкам і забезпечуючи, щоб усі елементи моделі відповідали встановленим правилам та вимогам.

Загалом, дослідження статичної та динамічної семантики для процесів моделювання в ASOME демонструє ефективність використання DMDSL для забезпечення структурованої і гнучкої архітектури системи, що підтримує точність і адаптивність коду, згенерованого на основі моделі.

РОЗДІЛ 3. ПРЕДСТАВЛЕННЯ МОДЕЛЕЙ ТА МЕТОДОЛОГІЇ ФОРМАЛЬНИХ СПЕЦИФІКАЦІЙ ПРОЦЕСУ ГЕНЕРАЦІЇ КОДУ

3.1. Дослідження формальної моделі

Спочатку розглянемо короткий виклад припущень семантики:

• Розглядається лише реалізація сховища за замовчуванням, створена ASOME. Це можливо, коли є лише один інтерфейс домену. Формальна модель тут не стосується реалізацій репозиторію та кількох доменних інтерфейсів. Запровадження налаштованих реалізацій репозиторію було б зручнішим із підходом зверху вниз, де спочатку можна визначити специфікації для кількох інтерфейсів домену.

• Невпорядкована семантика: реалізація ASOME використовує невпорядковану семантику та послідовності. Таким чином, допускаються дублікати, а елементи можуть бути впорядкованими (Sequence) або невпорядкованими (Bag). У певному випадку навігація по вихідних посиланнях певної асоціації повертає сумку. Bag – це невпорядкована послідовність елементів, яка допускає повторення. Текст передбачає невпорядковані асоціації.

• Орієнтація сховища: передбачається, що всі сутності зберігаються в репозиторіях, орієнтованих на посилання. Це спрощує операцію Retrieve(), оскільки екземпляр не дублюється, оновлення відбуваються негайно, а стан сховища не змінюється.

• Відношення узагальнення/спеціалізації не розглядаються.

Першим кроком тут є визначення структури моделі ASOME. Формальна модель DMDSL представлена як модель M: M є кортежем *<Entity, Association>*, де Entity — це набір типів сутностей, а Association — це набір асоціацій у моделі.

Нехай Entity представляє набір сутностей. Визначення елемента e є Entity це кортеж:

$$e = < C, M, D, N >$$

Цей елемент e є Entity – це кортеж властивостей: Constructability (C), Mutability (M), Deleteability (D) і Multiplicity (N).

Constructability (C) = True означає, що Entity є Constructable.

Конструктивність (C) = False означає, що сутність неконструйована.

Змінність (M) = True означає, що сутність можна редагувати.

Змінність (M) = False означає, що об’єкт не можна редагувати або незмінний.

Можливість видалення (D) = True означає, що сутність можна видалити.

Можливість видалення (D) = False означає, що сутність не видаляється або не знищується.

Кратність (N) = <мінімум, максимум> — це кратність сутності.

По-перше, необхідна властивість кінців асоціації (джерело/ціль):

$$AssociationEndProperty = < Cascade , Multiplicity >$$

Каскад є логічним значенням і представляє властивість каскадного видалення цього кінця асоціації. Кратність – це пара кратності (min, max).

Тепер нехай набір Association визначає асоціації в DMDSL. Визначення елемента асоціації є :

$$a = < source; target, SProperty , TProperty >$$

Тут a.source є Entity , а a.target є Entity. Крім того, SProperty є Association-EndProperty, та TProperty є AssociationEndProperty. Для довідки SProperty.Cascade представляє Source Cascade Delete (SCD), а TProperty.Multiplicity — цільову множинність асоціації.

Кратність N — це пара (мінімум, максимум), така що

$$Multiplicity = \{(min, max) \mid min >= 0 \wedge max >= min \wedge max > 0\}$$

Для перевірки DMDSL семантика моделі M буде визначена як перехідна система. Станом у цій системі є вміст сховищ і примірників, яких ще немає в сховищах, і цей стан також має вказувати на зв'язки між примірниками. Переходи відповідають операціям CRUD+A, які можна виконати в репозиторії. Наступні визначення припускають фіксовану модель M: сутності та асоціації в M не змінюються після створення екземпляра, і для моделі M існує система переходів, яка відповідає їй, яку визначає специфікація.

У динамічній семантиці сутність може бути створена. Нехай заданий екземпляр бути всесвітом усіх випадків, які можуть існувати. Кожен екземпляр x має тип об'єкта, з якого його створено. Таким чином, вид є властивістю екземпляра. Нехай тип : екземпляр ^ сутність бути функцією на екземплярі x такий, що тип (x) = y де y є Entity є сутністю екземпляра x. Ця функція заповнюється створенням екземпляра та видаленням екземпляра

Посилання є екземпляром асоціації. Посилання це мультинабір 1 , який фіксує всі посилання під час виконання. A link l є Link є кортежем:

$$l = < source; association; target >: Instance \; x \quad Association \; x \; Instance.$$

Відповідник семантиці сумки які представляють передбачувані специфікації, можуть бути два посилання p і q так, що p.s = q.s, p.a = q.a, p.t = q.t .

У системі переходу для моделі M зі стану s одна з операцій CRUD+A може здійснити перехід до наступного стану s' , якщо операція виконана успішно. Крім того, операція може завершитися помилкою через ряд можливих помилок, які не змінять вміст стану. Мета виходу властивість стану вказувати статус операції CRUD. Перехід, який вказує на невдалу операцію CRUD+A, має невдалий вихід, має мати ціль як s себе. Випадок

невдачі не означає, що перехід не можна здійснити. Перехід все ще можливий, але стан результату інший.

Виходи можуть бути представлені такими мітками у вигляді переліку:

• Entity_MultiplicityMaximum: якщо максимальна кратність сутності порушена, її екземпляри не можуть бути створені.

• Entity_MultiplicityMinimum: якщо мінімальна кратність сутності порушена, її екземпляри не можна видалити.

• Entity_Unconstructable: Спроба створити сутність, яку неможливо створити.

• Entity_Immutable: Спроба змінити/оновити сутність, яка початково є незмінною.

• Entity_Undestructable: Спроба видалити екземпляр сутності, оскільки її тип є не знищувальним.

• Entity_UnexpectedAssociation: Спроба створити посилання для екземпляра для певної асоціації, де сутність не є джерелом.

• Entity_MissingAssociation: Спроба створити екземпляр сутності, де необхідний зв’язок не надається.

• Association_SourceMaximum : максимальна множинність джерел асоціації порушена (під час створення або оновлення).

• Association_TargetMaximum: максимальна цільова кратність асоціації порушена (під час створення або оновлення).

• Association_TargetMinimum: мінімальна цільова кратність асоціації порушена (під час оновлення або видалення).

• Link_TargetNotInRepository: для посилання вихідний екземпляр знайдено в сховищі, а цільовий – ні.

• Instance_NotInRepository: Спроба оновити або видалити екземпляр, якого немає в сховищі .

• Instance_AlreadyInRepository: Спроба додати екземпляр до репозиторію після того, як його вже було додано.

3.2. Визначення початкових станів моделі

Модель DMDSL є дійсною моделлю, якщо вона відповідає всім статичним обмеженням, і завжди повинна існувати система переходу, яка відповідає дійсної моделі DMDSL. Щоб визначити перехідну систему, потрібен початковий стан (або набір початкових станів).

Базовим початковим станом для системи переходу є стан, у якому вміст наборів I, REPO та L є порожнім. Цей випадок можливий, лише якщо всі сутності в M мають мінімальну кратність 0. Однак зворотне не вірно: якщо всі сутності в моделі мають мінімальну кратність 0, все ще можливо заповнити вміст початковий стан, поки не порушується максимальна кратність. Тому початковий стан може мати > 0 екземплярів.

Для невпорядкованої семантики та статичних обмежень DMDSL, а також логічного факту, що будь-яка кількість екземплярів (нижче максимальної кратності) може бути створена до виконання, початковий стан для дійсної моделі M у специфікації DMDSL завжди існуватиме. У цьому тексті дійсний початковий стан системи переходів для моделі M визначає обмеження для початкового стану в термінах множинності об’єктів і асоціацій.

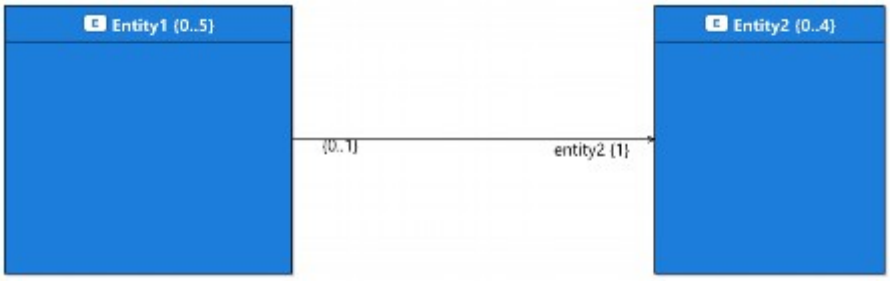

Рис. 3.1. Приклад моделі в DMDSL

Розглянемо рисунок 3.1 - для цієї моделі є кілька початкових станів. Один із них порожній вміст сховищ. Однак у цій моделі є ще 8 початкових

станів: репозиторій може мати 0 ... 5 екземплярів Entityl і 0 ... 4 екземпляри Entity2.

Отже, вимоги/обмеження початкового стану такі:

• необхідно дотримуватися множинності об’єктів;

• необхідно дотримуватися кратності асоціацій;

• посилання мають бути дійсними.

Практично користувач DMDSL вибирає вміст початкового стану. Таким чином, мета полягає в тому, щоб довести, що за умови дійсного початкового стану кожен перехід повинен вести до стану, де виконуються інваріанти. Якщо інваріанти порушено, це означає, що одна (чи більше) операцій CRUD+A не вказано правильно. Якщо операції CRUD+A указано правильно, тоді специфікація вважається дійсною.

Цей доказ можна виконати математично або за допомогою інструменту. Тут використовується інструмент Alloy, як описано в наступному розділі. Крім того, Alloy також надає інші переваги, такі як автоматичне створення або завершення вхідних моделей, а також створення тестових послідовностей. Таким чином, проблема, яку потрібно вирішити в Alloy, полягає в наступному: «За умови дійсного початкового стану для специфікації DMDSL, чи існує якась модель M і відповідна послідовність переходів, які порушують інваріанти узгодженості сховища?» Alloy Analyzer створює моделі для специфікації DMDSL і шукає моделі та послідовності операцій CRUD+A, які роблять інваріанти недійсними. Якщо така модель не знайдена, то специфікація вважається правильною.

3.3. Дослідження інструментів формальної специфікації

У цьому розділі семантика DMDSL визначена мовою специфікації Alloy. DMDSL вказано в Alloy з метою перевірки правильності операцій CRUD+A з точки зору узгодженості сховища. Використання інструментів специфікації для підтвердження правильності специфікацій має різні

переваги та недоліки. Практична перевага полягає в тому, що не потрібно доводити математичні теореми, оскільки замість цього використання Alloy дозволяє досліджувати модель і автоматично доводити властивості. Тому формальну специфікацію тепер перекладено на синтаксис Alloy.

Існують інструменти специфікації, такі як mCRL2 [12], Event-B [1], TLA+ [27], Z3 theorem prover [6]. Вимоги до програмного забезпечення перевірки:

• визначення вмісту станів у формі наборів або списків, а також поведінка систем переходу.

• підтримка параметричних переходів у системах переходів.

• простота використання зі знайомою нотацією теорії множин.

• перевірене використання та знайомство під час перевірки DSL у промислових середовищах.

Alloy як специфікаційний інструмент можна використовувати для перевірки специфікацій DMDSL. Alloy використовувався для перевірки моделей і мов. Можна визначати вміст сигнатур і статичну семантику, і параметричні предикати. Специфікація Alloy виражена в реляційній логіці, а синтаксис, як правило, простий.

Alloy — це декларативна формальна мова специфікації, натхненна мовою Z-специфікації та реляційним численням. Alloy використовується для опису структур і виконання обмеженого дослідження структури за допомогою Alloy Analyzer. Модель Alloy — це набір обмежень і рівнянь, описаних на наборі структур.

У моделі Alloy підписи визначають всесвіт моделі, і кожен підпис є набором. Кожен набір можна заповнити атомами — атоми генеруються Alloy, але їх множинність і співвідношення моделює користувач. Відношення в Alloy відображає підпис на один чи більше підписи. Його можна визначити як властивість сигнатури домену, а діапазон відношення може бути самим доменом (наприклад, ребро є відношенням графа та відображає кожну вершину в графі на 0 або більше вершин). Семантика Alloy підтримує

двійкові зв’язки нестандартно, а трійкові зв’язки можна реалізувати за допомогою бібліотеки. Відносини також мають множинність, як пояснюється пізніше. Підпис можна зробити абстрактним , щоб створити дочірні підписи, успадковані від абстрактного підпису.

Кожне твердження в моделі Alloy є булевим рівнянням. Факт може бути використаний для визначення обмежень на сигнатури в моделі (тобто рівнянь , які завжди будуть істинними). Предикат можна використовувати для визначення булевих рівнянь, на які можна посилатися за допомогою ідентифікатора, тобто предикати мають імена (тоді як факти не називаються). Твердження також можна використовувати для визначення булевих рівнянь і може містити посилання на предикати або інші твердження. Однак зворотне не вірно, і твердження не можна використовувати в команді запуску. На твердження можна посилатися лише за допомогою команди перевірки.

3.3.1. Виконання моделей Alloy

Alloy розглядає модель як кон’юнкцію булевих рівнянь, що можливо лише тому, що всі оператори Alloy є булевими рівняннями. Під час аналізу ці рівняння формуються у формулу SAT. SAT — це проблема визначення того, чи існує для заданої булевої формули набір присвоювань усім змінним у ній, щоб результат був TRUE . Навпаки, щоб знайти контекст, де формула має бути завжди істинною, формулу можна заперечити, а SAT можна застосувати для пошуку присвоєння змінним, де формула має значення FALSE . Загальновідомо, що проблема SAT в логіці предикатів є нерозв'язною .

Екземпляр моделі Alloy визначається як набір атомів сигнатур у заданих межах , які або задовольняють, або скасовують формулу SAT. Alloy — це інструмент дослідження обмеженої моделі, заснований на гіпотезі малого масштабу, Alloy досліджує лише моделі в заданих межах. Щоб знайти будь-який екземпляр, який задовольняє формулу SAT, використовується команда run. Щоб переконатися, що жоден екземпляр не робить модель недійсною, перевірте виконання команди. Чек, по суті, заперечує формулу

SAT і намагається знайти будь-який екземпляр, який її задовольняє. Таким чином, він перевіряє відсутність випадків, коли вхідні рівняння порушуються.

Користувач може захотіти перевірити модель на дійсні екземпляри або на (будь-які) недійсні екземпляри. Щоб виконати модель, або біг або перевірити використовується команда, яка вимагає рівнянь для перевірки та верхніх меж для моделі.

Виконання команди знаходить будь-який дійсний екземпляр для якого його вхідні рівняння мають значення TRUE. Якщо SAT не може бути задоволено, потрібно виконати вхідні рівняння визначаються несумісними, тобто не існує можливого екземпляра моделі, який задовольняє предикати в команді запуску. Чек команда знаходить будь-який недійсний екземпляр моделі, де вхідне рівняння має значення FALSE . Якщо тут не знайдено екземпляр, то твердження може бути дійсним для заданих меж. Можливо, вхідні рівняння мають суперечності. Однак такі випадки легко знайти і їх можна відкинути, тому що тут перевірте припиняється невиправдано швидко. Успішне завершення перевірки Команда не гарантує, що специфікація дійсна: вона гарантується лише в заданих межах. Однак це все ще створює впевненість щодо специфікацій через припущення невеликого контексту. Але це не гарантує справедливості тверджень необмеженого простору.

3.3.2. Практичне використання

Alloy ефективний для дослідження скінченних просторів станів для пошуку моделей, а також підходить для формальних специфікацій на основі стану. Інструменти, схожі на Alloy, це TLA+ [27], mCRL2 [12], Event-B [1] тощо. OCL схожий на Alloy щодо виразності. Хоча OCL має перший порядок логіки предикатів, Alloy має більш потужне реляційне числення та простіший синтаксис.

Моделювання систем переходу в Alloy непросте. Перехідні системи не можна виразити як першокласні сутності в Alloy, але їх потрібно моделювати вручну за допомогою бібліотеки Ordering. Це також згадується у відомих прикладах Alloy, гри River crossing та веб-атаки . В обох цих прикладах описується кінцевий автомат разом із предикатами, що визначають переходи/динамічну поведінку. Щоб дослідити впорядкування станів, перше, наступне та останнє ключові слова доступні в бібліотеці впорядкування . Задокументовано Alloy Java API , який можна використовувати для обробки згенерованих виходів в аналізаторі.

Лістинг 3.1. Специфікація графа в Alloy, перевірена на ациклічність

```
1  sig Node{
2      target_nodes: set Node    // relation
3  }{
4      target_nodes = { e: Edge | e.source = this }.target
5  }
6
7  sig Edge{
8      source: Node,
9      target: Node
10 }{
11     source != target
12 }
13
14 one sig Graph{
15     nodes: set Node,
16     edges: set Edge
17 }{
18     edges = Edge
19 }
20
21 pred NotCyclic{
22     all n: Node | n not in n.^target_nodes
23 }
24
25 check { NotCyclic } for 3
```

В лістингу 3.1 наведено короткий приклад, щоб показати синтаксис (і його неформальне значення) мови Alloy. У цьому прикладі визначено орієнтований граф. Цей граф має вузли та спрямовані ребра. Після підписів є фігурні дужки (рядки 4, 11) – вони називаються неявними фактами та застосовуються до самого підпису. Неявні факти застосовуються до кожного атома підпису (ключове слово this відноситься до atom) і корисні для визначення бінарних відносин (рядок 4). З рядка 4 набір target_nodes для

заданого атома Node x переглядає підпис Edge, де джерелом є this=x, і з цього набору витягує цільовий елемент, який має мати очікуваний тип набору Node, як у рядку 2 .

Сигнатура Edge (рядки 7-12) визначає ребро, а також забороняє самоцикли, тобто вузол може не мати ребра до себе. Підпис Graph не використовується під час перевірки моделі, але ключове слово one визначає, що під час дослідження моделі розглядається лише один графік. (В іншому випадку сигнатура графа досліджує всі перестановки вузлів і ребер. Для межі 3 є 3! = 6 графів, з яких 3 досліджуються)

Вважайте, що мета полягає в тому, щоб попросити Alloy підтвердити, що графік є ациклічним. За визначенням, вузол не повинен досягати будь-якої кількості ребер самого себе. Для цього в рядку 22 визначається предикат NotCyclic із оператором транзитивного замикання ". Він означає, що для всіх вузлів вузол n не повинен бути знайдений у транзитивному замиканні його набору target_nodes. У цьому випадку мета шукати будь-які недійсні моделі, тому команда check використовується в рядку 25 із верхньою межею підпису 3. Alloy досліджуватиме моделі для розмірів підпису від 0 до 3, але використовуючи ключове слово "точно" перед 3, лише підпис розмір 3 вважається. Запуск цього прикладу в Alloy створить контрприклад, як показано на рисунку 3.2.

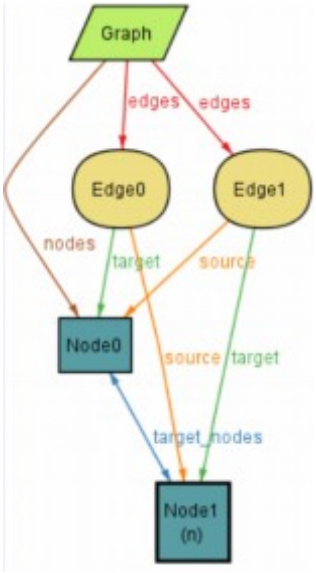

Рис. 3.2. Контрприклад для прикладу циклічності графа

Для усунення несправностей контрприкладу для перегляду інформації використовується функція оцінювача аналізатора. Стає зрозумілим, що порушення відбувається через те, що вузол 0 і вузол 1 вказують один на одного, як підтверджено за допомогою транзитивного замикання на рисунку 3.3. Треба зауважити, що третя команда використовує оператор ~. Таким чином, у цьому контрприкладі є цикл довжини 1.

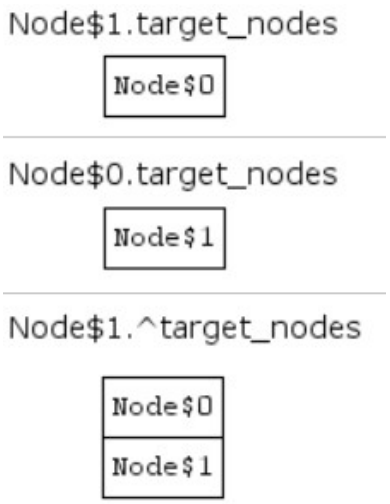

Рис. 3.3. Приклад: використання Alloy Analyzer для перевірки циклічності

3.3.3. Переваги і недоліки дослідження моделей за допомогою Alloy

Дослідження моделей за допомогою Alloy має низку практичних переваг:

• практика в інженерії програмного забезпечення: Alloy корисний для визначення специфікацій і для заданих меж, пошуку дійсних моделей за допомогою циклу команда. Якщо це вдається, специфікація демонструє бажані властивості.

• Alloy написаний на Java і доступний як один розповсюджуваний JAR, тому він не залежить від платформи. Синтаксис Alloy простий у вивченні, а твердження формальної логіки з теорією множин і логікою предикатів легко перекладаються в оператори Alloy.

• експресивність і сколемізація: мова Alloy визначає оператори для транзитивного замикання, ~ , для бінарних відносин. Це частина семантики логіки вищого порядку [17], яка включає транзитивне замикання, квантифікацію над множинами ступенів, аксіоми щодо дійсних чисел тощо, усі вони мають відношення до цього тексту.

Кількісну оцінку над наборами потужностей потрібно ввімкнути явно, налаштувавши параметр Skolemization 2-го порядку. У специфікації DMDSL транзитивне замикання використовується для обчислення циклічності асоціацій і для пошуку підключених екземплярів у наборі посилань.

• Хоча параметри за замовчуванням не дозволяють це, Alloy можна налаштувати для ввімкнення обмеженої рекурсії. Це також було експериментовано для формальної семантики DMDSL, але оператор транзитивного замикання забезпечує необхідну семантику та завжди закінчується тоді як семантика рекурсії цього не робить.

Існують різні обмеження щодо перевірки характеристик моделі за допомогою Alloy:

• точна довжина трас: щоб отримати точні результати за допомогою Alloy, важливо точно встановити межі для перевірки моделі. У бібліотеці Ordering система переходу повинна бути визначена для точної довжини. Можливо, що довга послідовність станів закінчується тупиком, і аналізатор не повідомляє про це.

• масштабованість: Alloy також страждає від вибуху простору станів, і залежно від складності моделі навіть збільшення простору станів, обмеженого 1, може призвести до значно довшого часу перевірки. Подібним чином, зменшення його на 1 може призвести до пропущеного підрахунку erexamples. Тому експериментування з межами зазвичай вимагається і трудомістко.

• у Alloy перевірка типу оцінюється глобально, тому назви атрибутів, зв’язків і предикатів потрібно визначати ретельно, щоб уникнути помилок розпізнавання області та типу. Подібні перешкоди реалізації можуть

ускладнити визначення множин і зв’язків, вимагаючи зусиль для вивчення нотації Alloy.

• практичним обмеженням реалізації з Alloy є дедукція невдалих умов: якщо твердження порушується та створюється контрприклад, яке рівняння спричинило невдачу. Аналізатор показує деяку пов’язану інформацію, але для складних моделей твердження в аналізаторі потрібно перевіряти вручну.

3.4. Представлення формальної специфікації із використанням інструменту Alloy

Враховуючи формальну специфікацію DMDSL, метою є перевірка правильності операцій CRUD+A щодо узгодженості сховища. Використання такого інструменту, як Alloy, для автоматичного дослідження моделі та перевірки узгодженості практичніше, ніж використання доведення теорем. Таким чином, тут формальна специфікація DMDSL, виражена в Alloy. Знаючи, що Alloy підтримує реляційну логіку, яка є сильнішою за логіку предикатів першого порядку та дозволяє використовувати транзитивний оператор закриття, формальну специфікацію можна перекласти за такою стратегією:

1. Перекладіть усі набори в підписи: сутність, асоціацію, екземпляр і посилання. Властивості Multiplicity та AssociationEndProperty також є підписами, які будуть властивостями Entity та Association. Для всіх цих підписів має бути забезпечено правильне оформлення .

2. Переведіть статичні обмеження (розділ 2.4) як факти. Ці факти повинні враховуватися Alloy Analyzer і завжди будуть вірними.

3. Потім визначте стан як підпис, щоб можна було моделювати його вміст (властивості). Оскільки метою є моделювання переходів, стан визначається як послідовність. Тому для його реалізації необхідна бібліотека впорядкування - без використання впорядкування атоми State є непересічними/виключними.

Бібліотека впорядкування дозволяє визначати послідовності між атомами і підпис. Властивості first, last, next упорядкування корисні для визначення переходів між елементами. Таким чином, система переходів може бути визначена та пройдена за допомогою бібліотеки впорядкування.

4. Перекладіть операції CRUD+A у предикати. Кожен предикат є набором булевих рівнянь, тому успішні операції можна визначити як предикати. У кожній операції CRUD+A зв’язок між даним станом і його наступним станом явно моделюється.

5. Щоб включити шляхи відмови, можна використати інший предикат. Цей предикат є логічною диз'юнкцією успішного предикату та кожної умови невдачі з відповідною поведінкою системи переходу.

6. Сліди фактів, які визначають переходи між двома станами s і s.next - використання наступного відношення можливе через бібліотеку впорядкування, застосовану до стану. Трасування визначає переходи між станами як операції CRUD+A.

7. Визначте умови початкового стану як факт. Враховуючи, що цей факт вимагає дійсного початкового стану, операції CRUD+A можна перевірити з точки зору дотримання інваріантів узгодженості сховища.

8. Перекладіть інваріанти узгодженості репозиторію в предикати над станом - параметром є стан, і всі сутності/асоціації повинні бути перевірені на дійсність їх множинності. Якщо цей предикат має значення true, тоді репозиторій є дійсним у стані.

9. Визначте очікуване виконання моделі. Використовується команда check{} із перевіркою меж моделі та точної довжини стану. Вхідні рівняння для перевірки {} є інваріанти узгодженості сховища. Таким чином, аналізатор Alloy може перевірити правильність переходів CRUD+A з дійсного початкового стану. Насправді, з достатньо чітко визначеними операціями CRUD+A ця перевірка в ідеалі не повинна давати контрприкладів.

3.4.1. Статична семантика DMDSL і обмеження правильного формування

Визначення статичної семантики DMDSL у Alloy є простим для наборів. Конструкції Multiplicity, Entity та Association, як зазначено в попередньому розділі, перекладаються як підписи. Відповідні властивості, такі як відношення цілей , були додані на основі їх призначення. Цілі — це відношення , що відображає сутність на сутності, яких вона може досягти за допомогою асоціацій, тому це відношення використовуватиметься в Alloy для визначення того, що моделі є ациклічними. Зверніть увагу на схожість із прикладом ациклічного графіка в лістингу 3.1 (Entity = Node, Association = Edge). Для забезпечення ациклічної моделі використовується транзитивний оператор замикання ~.

Лістинг 3.2. Формальна модель DMDSL у синтаксисі Alloy

```
sig Multiplicity{
    minimum: Int,
    maximum: Int
} {
    minimum >= 0 and maximum > 0 and maximum >= minimum
}

sig Entity {
    constructability: Bool,
    mutability: Bool,
    deleteability: Bool,
    multiplicity: one Multiplicity,
    targets: set Entity
} {
    targets = { a: Association | a.source = this }.target
}

sig AssociationEndProperty{
    cascade : Bool,
    multiplicity: Multiplicity
}

sig Association{
    source: Entity,
    target: Entity,
    sourceProperty: AssociationEndProperty,
    targetProperty: AssociationEndProperty
}
```

Для визначення статичних обмежень використовується наступний факт:

Лістинг 3.3. Статичні обмеження моделі DMDSL

```
1  fact constraints{
2      StaticConstraints
3  }
4
5  pred StaticConstraints{
6      no e: Entity | e in e.^associationTargets   // no cycles
7
8      all e: Entity {
9          (e.multiplicity.minimum = e.multiplicity.maximum) implies e.
               deleteability = False
10     }
11     // association properties: each line is a static constraint
12     all a: Association
13     {
14         a.sourceProperty.multiplicity.minimum = 0
15
16         a.target.deleteability = True implies (a.sourceProperty.cascade = True
               or a.targetProperty.multiplicity.minimum = 0)
17
18         a.sourceProperty.cascade = True implies (a.source.deleteability = True
               and a.source.multiplicity.minimum = 0)
19
20         a.sourceProperty.cascade = False implies a.source.mutability = True
21     }
22 }
```

Структури часу виконання, якими є екземпляр і посилання, перекладаються на Alloy, як зазначено нижче. Обмеження, визначені під підписами (у рядках 6,7,8,16,17), використовуються для забезпечення правильного оформлення підписів. В екземплярі зв'язки connectsSCD і connectsTCD використовуються під час видалення для побудови графіка підключених екземплярів (виключно для асоціацій із підтримкою SCD/TCD).

Лістинг 3.4. Сигнатури, пов'язані з часом виконання

```
1  sig Instance {
2      type: Entity,                // the type() function
3      connectsSCD: set Instance,
4      connectsTCD: set Instance
5  }{
6      type.constructability = True
7      connectsSCD = { L: Link | L.link_target = this and L.link_type.
           sourceProperty.cascade = True }.link_source
8      connectsTCD = { L: Link | L.link_source = this and L.link_type.
           targetProperty.cascade = True }.link_target
9  }
10
11 sig Link {
12     link_source: Instance,       // L.s
13     link_type: Association,      // L.a
14     link_target: Instance        // L.t
15 } {
16     link_source.type = link_type.source
17     link_target.type = link_type.target
18 }
```

Multiset Link з формальної специфікації природним чином перекладається на підпис Link (рядок 11), і це визначення в семантиці Alloy за замовчуванням дозволяє окремим посиланням мати однаковий вміст. Таким чином, семантика сумки легко фіксується Link.

3.4.2. Визначення стану моделі

У моделі Alloy стан визначається як сигнатура. Система переходів у Alloy кодується як послідовність станів, використовуючи бібліотеку впорядкування. Вміст стану тепер може бути визначено: екземпляри, репозиторії та зв'язки як властивість стану визначаються спочатку (рядки 4,5,6). Операція є простим індикатором операції, виконаної в переході з попереднього стану, а вихідне відношення представляє результат успіху або помилки. Факт для сигнатури State (рядок 10) визначає, що репозиторії є підмножиною екземплярів.

```
1  open util/ordering[State][language=Alloy, label=lst:AlloyInitState, caption={
       Definition of State in Alloy}]
2
3  sig State{
4      instances: set Instance,          // S.I
5      repositories: set Instance,       // S.REPO
6      links: set Link,                  // S.L
7      output: Output,       // output of CRUD operation of previous state
8      operation: Operation,             // CRUD operation of previous state
9  } {
10     repositories in instances
11 }
```

Щоб забезпечити дійсність початкового стану, використовується факт. Тут умови в початковому стані визначатимуть дійсний репозиторій, щоб поважалися сутності та множинності асоціацій. Зв'язки в початковому стані також примусово визначаються як дійсні.

Факт слідів є значним: він дозволяє визначати можливі переходи в системі переходів. З будь-якого стану s в системі точно один із предикатів create, add, update або delete може мати значення TRUE, щоб був здійснений перехід до наступного стану s0 = s:next. Видалення будь-якого одного з

рядків у слідів видаляє операцію CRUD+A з можливих переходів. Зауважте, що кожен стан у моделі Alloy може виконувати перехід CRUD. Це часто є вимогою в модельному тестуванні з системами переходів введення-виведення (IOTS) [24].

Лістинг 3.5. Визначення початкового стану за допомогою Alloy

```
fact traces
{
    all s: State - last, s': s.next
    {
        some e: Entity, targets: set Link  | create [s, s', e, targets]
        or some x: Instance  |  add[s, s', x]
        or some y: Instance, targets: {set Link - y} | update[s, s', y,
            targets]
        or some z: Instance  | delete[s, s', z]
    }
}

fact first_state{   // 'first' is provided by the ordering library, as the
    entry point of the ordering
    first.operation = Initialize
    first.output = Success

    Entity.multiplicity.minimum = {0} implies (no first.instances)

    all e: Entity | EntityMultiplicity[first, e]

    all a: Association  | AssociationMultiplicity[first, a]

    all p: first.links {
        p.link_source in first.instances and p.link_target in first.instances

        p.link_source in first.repositories implies p.link_target in first.
            repositories
    }
}
```

3.4.3. Виконання моделі

Щоб запустити модель Alloy, команда check використовується разом з інваріантами узгодженості сховища як вхідні дані. У цьому прикладі кількість підписів обмежена 3, а в системі переходу є 8 станів. Код тут представляє намір перевірити семантику операцій CRUD+A. Враховуючи систему переходів для деякої моделі m (яку визначає Alloy Analyzer), кожен перехід є однією з операцій CRUD+A. Команда check {Consistency} перевіряє, чи ці переходи не порушують інваріанти в будь-якому стані системи переходів. Таким чином, цей код використовується для відповіді на основну мету цієї роботи: чи семантика визначена правильно так, що

інваріанти не можуть бути порушені для будь-якої моделі m? Якщо команда перевірки {Consistency} не дає жодних контрприкладів, то передбачувана семантика вказана правильно. У цьому випадку не потрібні статичні обмеження, а також не потрібні уточнення операцій CRUD+A. Для кожного отриманого контрприкладу семантика уточнюється.

Лістинг 3.6. Специфікація DMDSL для перевірки моделі в Alloy

```
pred Invariants{
    all s: State {
        all e: Entity | EntityMultiplicity[s, e]
        all a: Association | AssociationMultiplicity[s, a]
        all p: s.links {
            p.link_source in s.instances and p.link_target in s.instances

            p.link_source in s.repositories implies p.link_target in s.
                repositories
        }
}

pred Consistency{
    Invariants
}

check { Consistency } for 3 but exactly 8 State
```

3.4.4. Використання Alloy для тестування моделі

Можливий більш заснований на моделі підхід до вивчення семантики DMDSL в Alloy. У цьому контексті можна вказати вхідні тестові моделі, а замість перевірки можна використовувати команду запуску . Як наслідок, це дозволяє тестувати специфікації DMDSL (як перехід система), для даної моделі, яка задовольняє деякі властивості. Властивості можна вказати в Alloy за допомогою предикатів, де до моделі можна застосувати один (або кілька) із наведеного нижче:

• Досліджуйте модель, лише якщо вона має такі властивості (model_test_properties), як кількість сутностей і асоціацій.

• Досліджуйте модель, лише якщо вона містить набір очікуваних операцій CRUD+A (model_test_traces): наприклад, щонайменше дві успішні операції видалення.

• Досліджуйте модель, лише якщо вона відповідає очікуваному кінцевому («останньому») стану з деякими властивостями: наприклад, кінцевий стан повинен мати принаймні 2 сховища та посилання.

За допомогою команди run можна перевірити, чи поєднання інваріантів і model_under_test призводить до дійсного екземпляра моделі. Таким чином, Alloy Analyzer може знайти тестові послідовності, які задовольняють усі умови, і ці послідовності можна перевести у виклики функцій над згенерованим кодом. Послідовність також представляє очікувану поведінку реалізації, її можна перевести в код C++ і виконати на реалізації. Нижче наведено приклад.

Лістинг 3.7. Тестування з підтримкою моделі для пошуку дійсних екземплярів за допомогою Alloy

```
pred model_under_test{
    model_test_properties
    model_test_traces
    model_test_last_state
}

pred model_test_traces{
    some s: State | s.operation = Create and s.next.output = Success
}

pred model_test_properties{
    #Entity >= 1   and  #Entity <= 4
    #Association >= 1 and #Association <= 2
    #Instance >= 1 and #Instance <= 5
    #Link >= 1 and #Link <= 5

    some p, q: Association | p.target = q.source
}

pred model_test_last_state{
    #(last.instances)  >=  5   // last state should have >4 instances
    #(last.prev.links - last.links )  >=  1    // delete 1 at least link in
       last state
}

pred Consistency{
    Invariants
}

run { Consistency and model_under_test } for 4 but exactly 8 State
```

Отже, представлено специфікацію DMDSL, наведену в Alloy. Є багато переваг використання Alloy: він може автоматично досліджувати моделі, можливі в специфікації, і генерувати тестові послідовності, контрприклади (в

теорії) мають невеликий контекст. Недоліком є те, що систему переходів потрібно моделювати вручну, крім того, довжина переходів має бути конкретною, а згенерований контрприклад потрібно перевіряти вручну для кожного твердження.

Визначення інваріантів узгодженості репозиторію дозволяє використовувати команду checkO та перевіряти відсутність стану, де порушено множинність сутності або асоціації. Визначаючи факт, який обмежує переходи між станами як одну з операцій CRUD+A, ця перевірка підтверджує, що операції CRUD+A не порушують узгодженість сховища. Крім того, можна також визначити екземпляри моделі, які слід протестувати за допомогою Alloy, і створити тестові послідовності на моделях, які в майбутньому можна буде використовувати для тестування на основі моделі.

3.5. Перевірка правильності виконання специфікацій

Семантика DMDSL була специфікована в Alloy. Ці специфікації тепер можуть бути перевірені на коректність. Для тестування специфікації було застосовано підхід де початковий стан не був примусово визначений, з метою знайти моделі, де відсутній початковий стан: це моделі, які слід інвалідувати від генерації коду шляхом додавання статичних обмежень.

У цьому аналізі специфікація DMDSL не зобов’язана мати початковий стан. Це дає змогу зрозуміти додаткові обмеження для оцінки дійсності моделей. Як тільки початковий стан може бути дійсним, динамічна семантика може бути перевірена. У цьому розділі представлені моделі без початкового стану.

Реалізація ASOME забороняє користувачеві створювати асоціації з тією самою Entity, але не запобігає циклам, подібним до того, що показано на рисунку 3.4 . Ця модель має дійсний початковий стан із 0 екземплярами, але немає наступного стану, оскільки на цій моделі неможливо виконати операцію CRUD+A. Це пояснюється тим, що для створення екземпляра

Entity0 потрібен рівно 1 екземпляр Entity1 після Association0, але для Entity1 потрібен екземпляр Entity0 після Association1.

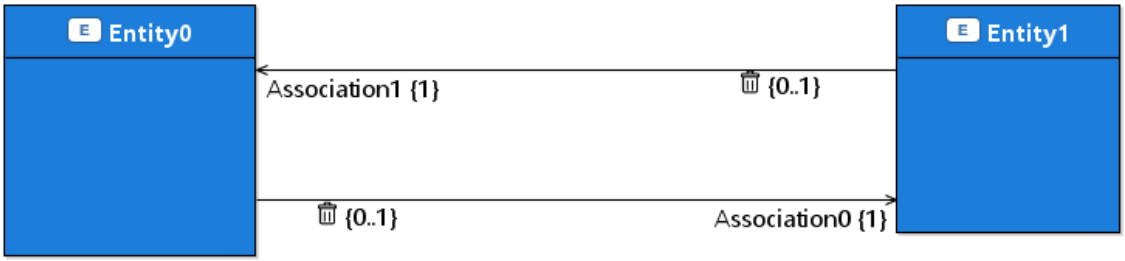

Рис. 3.4. Циклічні зв'язки в реалізації

Рішення: виявлення циклічності в реалізації DMDSL має бути розширено для виявлення циклів довжиною n.

Тепер розглянемо рисунок 3.5. Цей контрприклад не має дійсного початкового стану, оскільки для створення екземпляра Entity1 потрібно, щоб існував рівно 1 екземпляр Entity2 після асоціації 'entity2'. Однак Entity1 є Unconstructable, тому для розглянутих специфікацій DMDSL, які виключають семантику Realization, ніколи не буде можливо створити екземпляр Entity1. Таким чином, ніколи не буде можливо створити екземпляр EntityQ.

Рис. 3.5. Асоціація, яка ніколи не реалізується

Рішення: може бути надано нове обмеження: ціль асоціації завжди має бути Constructable, якщо вона має мінімальну кратність Q.

Наведена нижче модель має бути недійсною, оскільки початковий стан очікує принаймні 1 екземпляр Entityl через делегати конструктора, але Entity

має цільову Entity2, яка спочатку має екземпляри Q. Таким чином, початковий стан є недійсним, оскільки він не може задовольнити інваріанти множинності сутності.

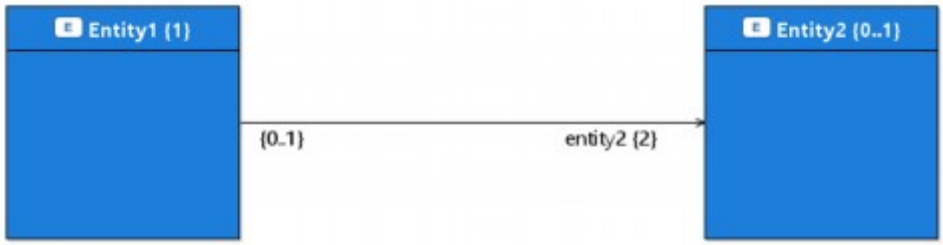

Рис. 3.6. недійсна модель через множинність

Рішення: у делегатах конструктора можна створювати екземпляри будь-якої сутності, тобто Constructable. Таким чином, початковий стан може мати >0 екземплярів для будь-якої сутності, яка є Constructable.

Деякі контрприклади були виявлені під час розуміння семантики успадкування DMDSL без формальної специфікації.

На рисунку 3.7 є асоціація від Entityl до Entity2. Існує також асоціація від Entity2 до Entity3. Ця модель поки що дійсна, але враховуйте відношення, що Entity3 є дочірнім елементом Entityl і успадковує асоціацію з Entity2. Це знову циклічна модель, яку перевірка повинна відхилити.

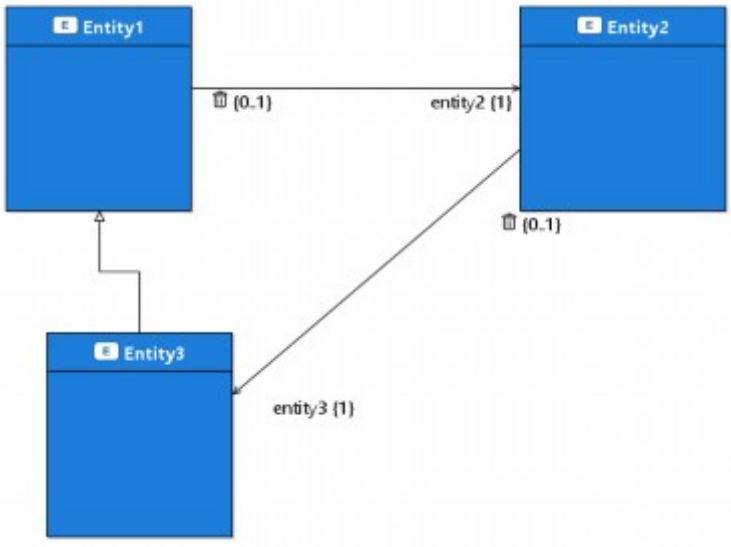

Рис. 3.7. Відношення спеціалізації створює цикли

Рішення: Виявлення циклічності має бути змінено, щоб перевірити наявність циклів після оцінки всіх відносин успадкування.

На рисунку 3.8 є асоціація від Entityl до Entity2. Існує також спеціалізація від Entityl до Entity2, тому Entity2 є спеціалізованим і успадковує всі асоціації Entityl. Це вводить циклічність, тому модель недійсна.

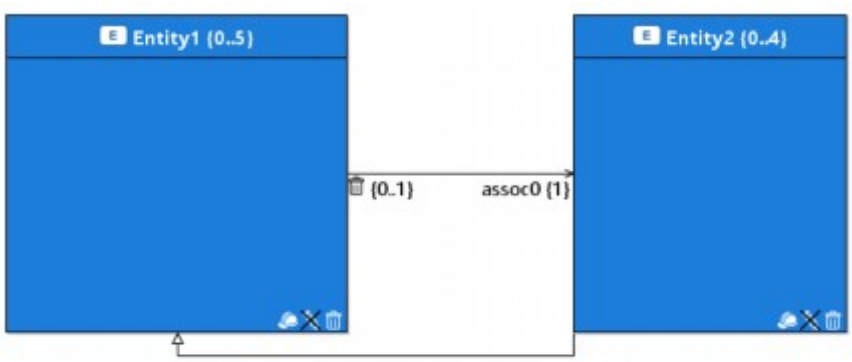

Рис. 3.8. Циклічна асоціація-спеціалізація

Рішення: Виявлення циклічності необхідно змінити, якщо враховується спеціалізація.

Експериментування з Alloy і вивчення контрприкладів є ретельним процесом, що спонукає до роздумів. Така вправа змушує розробників мови критикувати та розмірковувати над мовою: формальним синтаксисом моделей, обмеженнями правильного формування та динамічною семантикою. Аналіз змушує розробників мови інтерпретувати неінтуїтивні приклади моделей, згенеровані формальним інструментом перевірки, що призводить до вдосконалення семантики мови та зменшення помилок.

Висновки до розділу

У цьому розділі було представлено моделі та методології формальних специфікацій процесу генерації коду на основі доменно-специфічних

моделей, зокрема на прикладі DMDSL (Domain-Interface Modeling Domain Specific Language). У процесі дослідження було задокументовано ключові елементи формальної специфікації DMDSL, що дозволяють моделювати та перевіряти коректність основних операцій CRUD+A (створення, читання, оновлення, видалення та додавання) в контексті узгодженості сховища.

Основним підходом до розробки специфікації було початкове визначення інтерфейсів, які включають сутності та їхні асоціації, а також відповідні операції CRUD+A, що дозволяють реалізувати основні процеси управління даними. Важливою складовою дослідження є введення синтаксису та наборів конструкцій для представлення семантики DMDSL, що забезпечує гнучкість і точність при розробці моделей.

Важливим аспектом цього дослідження стала перевірка узгодженості сховища, де було досліджено коректність виконання операцій CRUD+A в межах формальної специфікації. Це дозволило забезпечити гарантії цілісності даних та їхнього коректного зберігання й обробки в системі.

Отже, представлена специфікація DMDSL на основі формальних методів сприяє покращенню якості моделей та забезпечує надійність і точність процесу генерації коду.

ВИСНОВКИ

У магістерській роботі проведено ґрунтовне дослідження методології формальних специфікацій для процесу генерації коду, що базується на доменно-специфічних моделях. Розроблено та проаналізовано інструменти для формальної верифікації семантики моделей, зокрема мову Alloy і систему AlloyAnalyzer, що дозволяють перевіряти коректність операцій CRUD+A у системі з точки зору узгодженості даних і якості коду.

Перший розділ роботи було присвячено теоретичному огляду предметної області, включаючи моделювання програмного забезпечення на основі доменно-специфічних мов, таких як DMDSL. Описано концепцію тестування на основі моделей та розглянуто актуальні інструменти для моделювання, що дозволяють підвищити якість та надійність систем за допомогою формальної специфікації.

Другий розділ досліджував статичну та динамічну семантику в рамках середовища моделювання ASOME, що забезпечує гнучкість та розширюваність у розробці програмних систем. Було визначено та описано специфікацію різних елементів моделі, включаючи сутності, асоціації та операції, що забезпечують інтерактивність та можливість проведення CRUD+A операцій у моделі.

Третій розділ присвячено формальній специфікації процесу генерації коду за допомогою інструменту Alloy. Розглянуто синтаксис мови Alloy для моделювання та перевірки DMDSL, що дозволяє формально описати як статичні, так і динамічні аспекти моделі. Описано процес виконання та тестування моделі на відповідність специфікації, що дозволяє визначити та усунути можливі помилки у розробці програмного забезпечення.

Для моделювання та перевірки формальних специфікацій було застосовано мову Alloy, яка дозволила провести валідацію моделі та дослідити можливі варіанти її виконання. Розділ також розкрив переваги та

недоліки використання Alloy для формальної верифікації, що допомогло уточнити процес моделювання.

Таким чином, результати проведеного дослідження підтвердили, що використання формальних методів, зокрема мови Alloy, для опису процесу генерації коду дозволяє покращити надійність і точність моделей, що використовуються в сучасних програмних системах. Представлена методологія дозволяє не лише верифікувати коректність моделі, а й забезпечити її відповідність вимогам користувача та узгодженість даних. Практичне застосування таких підходів може значно підвищити ефективність процесів розробки та тестування програмного забезпечення.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Jean-Raymond Abrial. Modeling in Event-B: system and software engineering. Cambridge University Press, 2010.
2. Axel Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 266–270. Springer, 2010.
3. Jesús Sánchez Cuadrado and Jesús García Molina. Building domain-specific languages for model-driven development. IEEE Software, 24(5):48–55, Sep. 2007.
4. Alcino Cunha, Ana Garis, and Daniel Riesco. Translating between alloy specifications and uml class diagrams annotated with ocl. Software & Systems Modeling, 14(1):5–25, 2015.
5. Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, volume 45, pages 1–17. USA, 2003.
6. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In International conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 337–340. Springer, 2008.
7. Doron Drusinsky. Modeling and verification using UML statecharts: a working guide to reactive system design, Runtime Monitoring and Execution-based Model Checking. Elsevier, 2011.
8. Ferhat Erata, Arda Goknil, Ivan Kurtev, and Bedir Tekinerdogan. Alloyinecore: Embedding of first-order relational logic into meta-object facility for automated model reasoning. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, page 920–923. Association for Computing Machinery, 2018.

9. Mark Frenken, Tim AC Willemse, Louis van Gool Océ, Olav Bunte, and Jasper Denkers. Code generation and model-based testing in context of oil. Master's thesis, Technische Universiteit Eindhoven, 2019.
10. Loïc Gammaitoni, Pierre Kelsen, and Christian Glodt. Designing languages using lightning. In Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, page 77–82, New York, NY, USA, 2015. Association for Computing Machinery.
11. Richard C Gronback. Eclipse modeling project: a domain-specific language (DSL) toolkit. Pearson Education, 2009.
12. Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck VanWeerdenburg. The formal specification language mcrl2. In Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.
13. David Harel. Statecharts: A visual formalism for complex systems. Science of computer programming, 8(3):231–274, 2007.
14. John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In Proceedings of the 33rd International Conference on Software Engineering, pages 633–642, 2011.
15. Daniel Jackson. Software Abstractions: logic, language, and analysis. MIT press, 2012.
16. Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science, 152:125–142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).
17. Dale A Miller and Gopalan Nadathur. Higher-order logic programming. In International Conference on Logic Programming, pages 448–462. Springer, 1986.
18. Rocco Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In Semantics of Systems of Concurrent Processes, pages 407–419. Springer Berlin Heidelberg, 1990.

19. Douglas C Schmidt. Model-driven engineering. Computer-IEEE Computer Society, 39(2):25, 2006.
20. Seyyed MA Shah, Kyriakos Anastasakis, and Behzad Bordbar. From uml to alloy and back again. In International Conference on Model Driven Engineering Languages and Systems, pages 158–171. Springer, 2009.
21. Marten Sijtema, Axel Belinfante, MIA Stoelinga, and Lawrence Marinelli. Experiences with formal engineering: Model-based specification, implementation and testing of a software bus at neopost. Science of computer programming, 80:188–209, 2014.
22. Terje Sivertsen. State-based and transition-based specifications, Feb 2001.
23. Ulyana Tikhonova. Reusable specification templates for defining dynamic semantics of DSLs. Software & Systems Modeling, 18(1):691–720, March 2017.
24. Jan Tretmans. Model based testing with labelled transition systems. In Formal methods and testing, pages 1–38. Springer, 2008.
25. Axel van Lamsweerde. Formal specification. In Proceedings of the conference on The future of Software engineering - ICSE. ACM Press, 2000.
26. Jos BWarmer and Anneke G Kleppe. The object constraint language: getting your models ready for MDA. Addison-Wesley Professional, 2003.
27. Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In Advanced Research Working Conference on Correct Hardware Design and Verification Methods, pages 54–66. Springer, 1999.
28. Abrial, J.-R. (2005). The B-Book: Assigning Programs to Meanings. Cambridge University Press.
29. Back, R.-J. (2007). Refinement Calculus: A Systematic Introduction. Springer.
30. Bowen, J. P., & Hinchey, M. G. (2006). Ten Commandments of Formal Methods. IEEE Computer.
31. Broy, M. (2010). Formal Methods in Software Engineering. Springer.

32. Butler, M. J., & Hartel, P. (2011). Formal Specification and Documentation using Z. McGraw-Hill.
33. Clavel, M., & Eker, S. (2007). Maude Manual, Version 2.3. SRI International.
34. Czarnecki, K., & Eisenecker, U. W. (2005). Generative Programming: Methods, Tools, and Applications. Addison-Wesley.
35. Fiadeiro, J. L. (2004). Categories for Software Engineering. Springer.
36. Goguen, J. A., & Burstall, R. M. (1992). Institutions: Abstract Model Theory for Specification and Programming. Journal of the ACM.
37. Goldsmith, S., & Jagannathan, S. (2009). Model-Driven Software Engineering for High-Assurance Applications. Springer.
38. Guttag, J. V., & Horning, J. J. (1993). Larch: Languages and Tools for Formal Specification. Springer.
39. Hehner, E. C. R. (2006). A Practical Theory of Programming. Springer.
40. Hoare, C. A. R. (1985). Communicating Sequential Processes. Prentice Hall.
41. Jackson, D. (2006). Software Abstractions: Logic, Language, and Analysis. MIT Press.
42. Jones, C. B. (1990). Systematic Software Development using VDM. Prentice Hall.
43. Kappel, G., Pröll, B., Reich, S., & Retschitzegger, W. (2005). Web Engineering: The Discipline of Systematic Development of Web Applications. John Wiley & Sons.
44. Kiczales, G., Lamping, J., Lopes, C. V., Maeda, C., & Murphy, G. (1997). Aspect-Oriented Programming. Springer.
45. Knapp, A., & Wirsing, M. (2003). Formal Methods for Software Architectures. Springer.
46. Koch, N., & Kraus, A. (2006). The UML-based Web Engineering Approach to Model-driven Web Application Development. Springer.
47. Meyer, B. (1997). Object-Oriented Software Construction. Prentice Hall.
48. Morgan, C. (1990). Programming from Specifications. Prentice Hall.

49. Nuseibeh, B., & Easterbrook, S. (2000). Requirements Engineering: A Roadmap. ICSE Proceedings.
50. Pnueli, A., & Manna, Z. (1992). The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer.
51. Roscoe, A. W. (1997). The Theory and Practice of Concurrency. Prentice Hall.
52. Stepney, S., Cooper, D., & Woodcock, J. (2006). More Challenges in Engineering Informatics using Formal Methods. IEEE Computer Society.
53. Wirsing, M., & Knapp, A. (2003). Formal Methods for Software Architectures. Springer.