МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХЕРСОНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних наук, фізики та математики

Кафедра комп'ютерних наук та програмної інженерії

ВАЛІДАЦІЯ ФОРМУЛ В СИСТЕМІ ІНСЕРЦІЙНОГО МОДЕЛЮВАННЯ.

Кваліфікаційна робота (проект)

на здобуття ступеня вищої освіти «магістр»

Виконав: здобувач 5 курсу 241М групи

Спеціальності 121 Інженерія програмного забезпечення

Освітньо-професійної програми Інженерія програмного забезпечення

Дубіна Владислав Геннадійович

Керівник: доктор фізико-математичних наук, професор Песчаненко Володимир Сергійович

Рецезент: Тарасіч Ю.Г., докторант інституту кібернетики імені В.М. Глушкова НАН України, керівник компанії «Garuda.AI»

Херсон – Івано-Франківськ – 2023

Вступ

РОЗДІЛ 1

Огляд процесу перевірки формул для системи моделювання вставок та алгебраїчної віртуальної машини

1.1 Досвід у використанні систем "AVM" та "IMS" у сучасному суспільстві

1.2 Огляд систем моделювання вставок

1.2.1 Система моделювання вставок (IMS)

1.2.2 Алгебраїчна віртуальна машина (AVM)

1.2.3 PE Litsoft Model Creator

РОЗДІЛ 2

Валідація формул

2.1 Параметризований базовий протокол

2.2 Валідація в Model Creator

2.2.3 Дерево вузлів.

2.2.4 Синтаксичні правила.

2.2.3 Типізація.

2.3 Валідація параметризованих протоколів

ВИСНОВКИ

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Вступ

Актуальність теми

Важливість точності та надійності моделей вставок: Системи моделювання вставок (IMS) використовуються в різних галузях, включаючи науку, інженерію та бізнес-аналітику. Точність та надійність моделей є критичними, оскільки на їхніх результатах може базуватися прийняття важливих рішень. Тому валідація формул у системі IMS є актуальною задачею для забезпечення коректності результатів моделювання. Великі системи IMS можуть містити сотні або навіть тисячі формул, що ускладнює їхню валідацію вручну. Внаслідок цього, автоматизований підхід до валідації стає вкрай важливим для забезпечення ефективності та продуктивності роботи з IMS. Системи моделювання, такі як AVM, стають все більш популярними завдяки своїм можливостям та потенціалу. З цим зростанням популярності збільшується і важливість валідації, оскільки залежність від них зростає, а разом з цим і ризик некоректності введених формул. Оскільки IMS і AVM розвиваються та вдосконалюються, потрібно постійно адаптувати та розробляти інструменти для валідації формул у великих системах. Це створює попит на дослідження та розробку нових методів та підходів до валідації для забезпечення актуальності теми.

Мета і завдання

Об'єкт дослідження - Система інсерційного моделювання (IMS), зокрема Algebraic Virtual Machine (AVM) компанії "PE Litsoft".

Предмет дослідження - проблеми та особливості валідації формул в цій системі, а також розробка валідатора, спрямованого на вирішення цих проблем.

Мета кваліфікаційної роботи полягає в розгляді проблем та особливостей валідації формул для системи моделювання вставок (IMS), зокрема для системи Algebraic Virtual Machine (AVM) компанії

“PE Litsoft". Огляд основних кроків, необхідні для розробки валідатора, спрямованого на перевірку коректності введених формул в даній системі.

Відповідно до мети можна визначети основні *завдання* роботи:

- Провести аналіз синтаксису та структури формул в системі IMS.

- Розробити алгоритм валідації формул, який враховує особливості IMS та AVM.
    - Реалізувати валідатор за допомогою мов програмування JavaScript та TypeScript на базі операційної системи Ubuntu.
    - Визначити та розділити аргументи формул, знайти їхню ініціалізацію та типи.
    - Перевірити правильність передачі аргументів у формулах та знайти помилки у коді поведінкових моделей.

- Провести тестування розробленого валідатора на реальних даних та інтегрувати його в систему IMS.

Апробація результатів дослідження

- P4Testgen: An Extensible Test Oracle For P4 / F. Ruffy, J. Liu, P. Kotikalapudi, V. Havel, R. Sherwood, V. Dubina, V. Peschanenko, N. Foster, A. Sivaraman // arXiv:2211.15300 [cs.NI], https://doi.org/10.48550/arXiv.2211.15300

- P4Testgen: An Extensible Test Oracle For P4-16 / F. Ruffy, J. Liu, P. Kotikalapudi, V. Havel, H. Tavante, R. Sherwood, V. Dubina, V. Peschanenko, A. Sivaraman, N. Foster // ACM SIGCOMM '23: Proceedings of the ACM SIGCOMM 2023 Conference, September 2023, Pages 136–151, doi: https://doi.org/10.1145/3603269.3604834

РОЗДІЛ 1

Огляд процесу перевірки формул для системи моделювання вставок та алгебраїчної віртуальної машини

1.1 Досвід у використанні систем "AVM" та "IMS" у сучасному суспільстві

Алгебраїчна віртуальна машина (AVM) та система моделювання вставок (IMS) стали важливою складовою сучасного технологічного ландшафту і мають значний вплив на різні галузі, включаючи інженерію, інформаційні технології та науку. Використання цих систем виявилися деякі ключові аспекти:

1. Моделювання та аналіз систем: Система моделювання вставок (AVM) надає потужний інструмент для створення моделей складних систем. AVM використовують для моделювання та аналізу процесів в різноманітних інженерних проектах. Це дозволяє більш ефективно розуміти, як різні компоненти взаємодіють між собою і як можна оптимізувати їх роботу.

2. Підвищення ефективності роботи: Використання IMS спростило процес розробки та тестування програмного забезпечення. Впровадження IMS у проектах з розробки програмного забезпечення для автоматизації бізнес-процесів дозволяє значно зменшити час розробки та підвищити якість нашого продукту.

3. Оптимізація ресурсів: Використання AVM та IMS, дає змогу оптимізувати використання ресурсів у проектах. Це включає в себе більш раціональне використання обчислювальних потужностей та зменшення витрат.

4. Забезпечення безпеки та надійності: IMS допомагає виявляти потенційні проблеми та вразливості у програмному забезпеченні та системах. Це сприяє забезпеченню безпеки та надійності наших продуктів та процесів.

У підсумку, досвід використання систем AVM та IMS довів їхню важливість у сучасному технологічному середовищі та їхню спроможність покращувати ефективність, надійність і безпеку наших проектів і систем.

1.2 Огляд систем моделювання вставок

Для того, щоб відповідно зрозуміти проблему перевірки формул для системи моделювання вставок, зокрема алгебраїчної віртуальної машини (AVM), необхідно насамперед докладніше дізнатися, що представляють собою ці системи та як вони працюють. Відсутність чіткого розуміння принципів роботи цих систем може призвести до непорозумінь та недоліків у перевірці формул, яка є важливою частиною процесу моделювання.

Додатково, ми детальніше розглянемо "Model Creator" – веб-застосунок, що використовується користувачами для створення моделей в рамках алгебраїчної віртуальної машини, і який також є інтелектуальною власністю компанії "PE Litsoft". Оскільки "Model Creator" є головним інструментом для створення моделей для AVM, саме в ньому проводиться перевірка формул. Він надає користувачам широкий спектр інструментів та можливостей для створення та редагування моделей, що робить його важливою частиною процесу валідації та оптимізації формул.

1.2.1 Система моделювання вставок (IMS)

Моделювання вставок - це передова технологія проектування систем, яка базується на теорії взаємодії агентів та середовища. Основна ідея полягає в тому, щоб створити систему, де окремі складові (агенти) взаємодіють зі своєю оточуючою середовищем, що може бути уявлено як площина або простір, на якому ці агенти діють.

Однією з головних мет цієї технології є об'єднання різних моделей взаємодії та обчислень. Це означає, що ви можете використовувати різні математичні або обчислювальні моделі (наприклад, CCS, CSP, π-калькуль, мобільні амбієнти тощо) для аналізу імітаційної системи вставок.

Останніми роками цей метод отримав широке застосування у верифікації вимог до розподілених паралельних систем у різних галузях, таких як телекомунікації, телематика, розподілені обчислення та інші. Він дозволяє аналізувати та вдосконалювати взаємодію агентів та їхнє спільне функціонування у складних системах.

У моделюванні вставок існують два основних типи суб'єктів: агенти і середовища. Агенти - це активні компоненти, які мають здатність взаємодіяти і змінювати свій стан. Середовище - це пасивна область, на якій діють агенти, і вона може бути описана як двовимірна площина чи простір.

Особливістю середовища в моделюванні вставок є наявність функції вставки. Ця функція дозволяє додавати агентів до середовища та визначати їхню взаємодію з оточуючими об'єктами. В результаті ця технологія надає можливість композиції та характеризації поведінки

середовища з доданими агентами для вивчення та оптимізації процесів в різних областях.

Моделювання вставок є потужним інструментом для аналізу та проектування складних систем, особливо тих, де важлива взаємодія між окремими компонентами або агентами. Основна ідея полягає в тому, щоб агенти, які можуть бути віртуальними сутностями або об'єктами в реальному світі, взаємодіяли між собою та з середовищем, у якому вони діють. Ця взаємодія може включати обмін інформацією, зміну стану агентів та вплив на оточуючі об'єкти.

Один із ключових аспектів моделювання вставок - це можливість об'єднувати різні моделі взаємодії та обчислень. Наприклад, ви можете використовувати математичні моделі (наприклад, π-калькуль) разом з імітаційними або агентно-орієнтованими моделями для дослідження взаємодії агентів у складних системах.

Застосування моделювання вставок в останні роки розширилося на різні галузі, включаючи телекомунікації, де важлива ефективна робота мережі та взаємодія різних пристроїв; телематику, де моделюються системи автомобільної безпеки та навігації; розподілене обчислення, де аналізуються взаємодії між обчислювальними вузлами та багато інших галузей.

Основною метою моделювання вставок є досягнення комфорту, ефективного використання ресурсів та підвищення безпеки в системах, де важлива взаємодія та співробітництво між різними суб'єктами. Такий підхід допомагає уникнути помилок, оптимізувати процеси та забезпечити надійність систем.[1, 2, 3]

1.2.2 Алгебраїчна віртуальна машина (AVM)

Алгебраїчна віртуальна машина (АВМ) - це концептуальна модель обчислювальної системи, яка базується на алгебрі поведінки і теорії взаємодії агентів та середовища. Ця концепція дозволяє моделювати та аналізувати взаємодію агентів в складних імітаційних середовищах. Основні компоненти АВМ включають в себе агентів, середовище, атрибути та поведінку.

Основні поняття та складові АВМ:

- **Агенти**: Агенти - це сутності або об'єкти, які мають здатність змінювати свій стан та взаємодіяти з іншими агентами та середовищем. Кожен агент складається з двох основних компонентів:
  - *Інформація*: Інформація визначає стан агента і представлена значеннями його атрибутів. Це дані, які агент може зберігати та використовувати для прийняття рішень.
  - *Поведінка*: Поведінка представляє собою набір можливих сценаріїв або дій, які агент може виконувати в різних ситуаціях. Це визначає, як агент реагує на події та взаємодіє з оточуючими сутностями.

- **Середовище**: Середовище - це контекст, в якому агенти діють. Воно може бути уявлено як площина, простір або конкретна система, де розгортаються імітаційні процеси. Середовище може мати свої атрибути та поведінку, які впливають на агентів та моделюють умови взаємодії.

- **Атрибути**: Атрибути - це характеристики агентів або середовища, які визначають їхні властивості та стан. Агенти можуть мати власні атрибути, які використовуються для зберігання даних та взаємодії з іншими агентами. Середовище також може мати свої атрибути, які відображають його характеристики

- **Поведінка середовища**: Поведінка середовища включає в себе набір правил і сценаріїв, які визначають, як середовище реагує на дії агентів та події в системі. Це важливо для моделювання взаємодії між агентами та оточуючим середовищем.

Алгебраїчна віртуальна машина використовує ці компоненти для моделювання та аналізу різних сценаріїв та взаємодій в системі. Вона може бути використана для вивчення поведінки агентів в різних умовах, оптимізації взаємодії між ними та вирішення проблем, пов'язаних з імітацією та моделюванням складних систем.[4, 5, 6]

1.2.3 PE Litsoft Model Creator

Model Creator - це програмне середовище, спеціально розроблене для створення моделей в системі моделювання вставок (Insertion Modeling). Він є інструментом, призначеним для аналізу та моделювання різних процесів і взаємодій в складних системах. Model Creator забезпечує користувачів інструментами і можливостями для створення, редагування та аналізу моделей в контексті вставок.

Основні характеристики Model Creator:

- **Розробка моделей**: Основна функція Model Creator полягає у розробці моделей для системи моделювання вставок. Він допомагає користувачам створювати складні моделі, які описують різні аспекти системи агентів та їх взаємодій.

- **Широкий набір інструментів**: Model Creator постачається з різними інструментами, які полегшують процес створення моделей. Ці інструменти можуть включати графічний редактор, текстовий редактор, інструменти для роботи з даними, а також можливості візуалізації даних.

- **Гнучкість і інтуїтивність**: Особливістю Model Creator є його гнучкість та інтуїтивний інтерфейс(див. рис.1.1). Він дозволяє користувачам легко створювати, редагувати та аналізувати моделі, надаючи широкі можливості для налаштування і оптимізації проекту.

- **Візуалізація**: Model Creator може надавати інструменти візуалізації даних і результатів моделювання, що дозволяє користувачам краще розуміти та аналізувати роботу їхніх моделей.

Загалом, Model Creator є потужним інструментом для інженерів, дослідників і аналітиків, які займаються моделюванням та аналізом складних систем. Він спрощує процес створення моделей і допомагає вивчати взаємодію агентів та середовища у різних умовах.

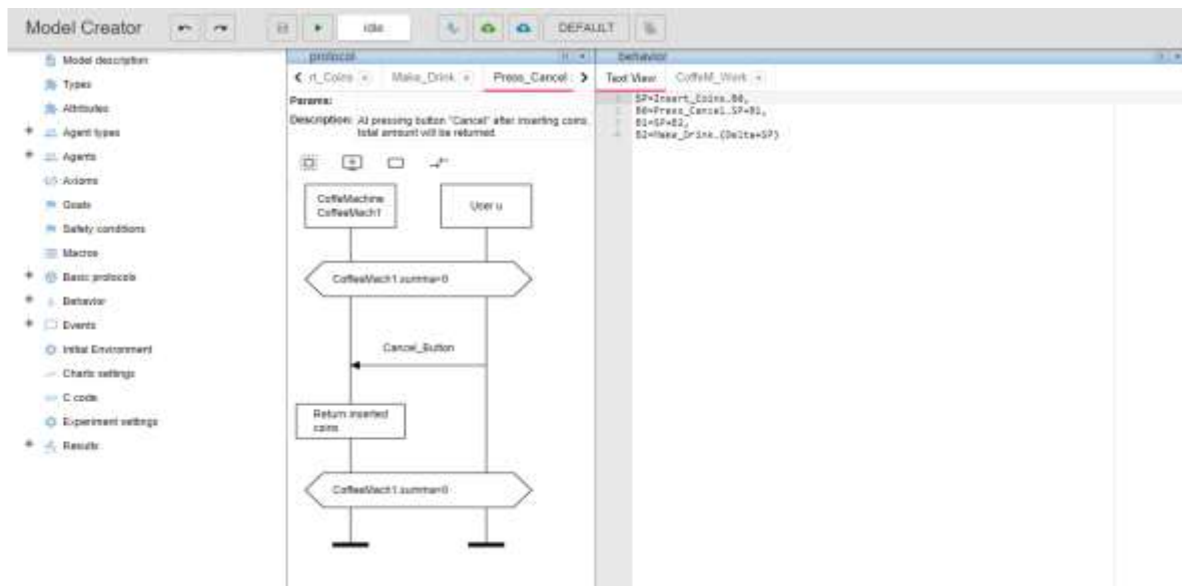

Рис. 1.1 - Зображення стандартної сторінки Model Creator з демонстраційною моделлю.

Model Creator має валідатор, який відображає помилки у написанні моделі, а також є консоль, в якій буде відображатися інформація про стан та процес виконання моделі (див. рис.1. 2).

console

Console Find Errors Debug

```
Saving project...
Saved!
connected
task compiling...
run task, 209
Loading types-done
Loading attributes-done
Loading agent types-done
Loading intial state-
done
Loading agents-done
Translation behavior -
done
[Tue Apr 11 18:20:37 2023]:[aplan2c.cpp,load_action,787]:[INF]:[J2A]:Loading action 'Press_Cancel'
[Tue Apr 11 18:20:37 2023]:[aplan2c.cpp,load_action,1058]:[INF]:[J2A]:done
[Tue Apr 11 18:20:37 2023]:[aplan2c.cpp,load_action,787]:[INF]:[J2A]:Loading action 'Insert_Coins'
[Tue Apr 11 18:20:37 2023]:[aplan2c.cpp,load_action,1058]:[INF]:[J2A]:done
[Tue Apr 11 18:20:37 2023]:[aplan2c.cpp,load_action,787]:[INF]:[J2A]:Loading action 'Make_Drink'
[Tue Apr 11 18:20:37 2023]:[aplan2c.cpp,load_action,1058]:[INF]:[J2A]:done
Loading events-done
done
done
```

Рис. 1.2 - Частина консолі з демонстраційними даними

Використовуючи Model Creator, ви отримуєте можливість проводити виконання моделі крок за кроком. Це означає, що ви можете послідовно виконувати кожен етап моделювання, спостерігаючи, як він розвивається. Ця можливість надає чітке розуміння того, як працює модель та де можуть бути помилки.

Після того як модель була запущена, ви можете переглядати результати, які включають в себе дані, графіки та багато іншої інформації. Це допомагає вам аналізувати результати моделювання та отримувати інсайти щодо роботи вашої моделі(див. рис.1. 3).

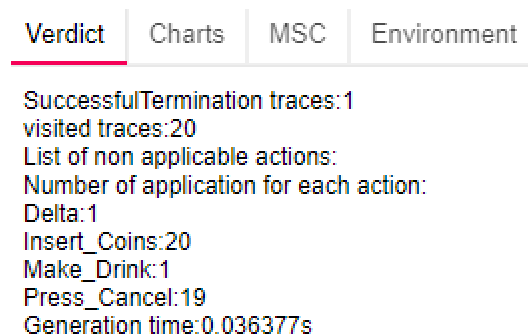

Рис. 1.3 - Блок із результатами моделі, заповненими демонстраційними даними

Отже, Model Creator дозволяє вам не лише створювати моделі, але й систематично виконувати їх, щоб легше зрозуміти їх роботу та виявляти можливі проблеми. Після виконання моделі, ви можете аналізувати результати, включаючи графіки та інші дані.

РОЗДІЛ 2

Валідація формул

Щоб розпочати розробку модуля валідації формул першочергово необхідно ознайомитися з структурою самого сервісу, тобто Model Creator. Сервіс , як і всі сучасні додатки, має модульну структуру що надає такі переваги як :

- **Легка розширюваність**: Завдяки модульній структурі можна додавати нові функції та можливості до додатку, не змінюючи весь код. Кожен модуль може бути розроблений окремо і підключений до додатку за необхідності
- **Покращена обслуговуваність**: Кожен модуль може бути підтримуваним і оновлюваним окремо. Це полегшує виявлення і виправлення помилок та забезпечує більш просту підтримку додатку.

- **Зменшення ризику помилок**: Використання модульної структури може допомогти зменшити ризик виникнення помилок. Кожен модуль може бути ретельно протестованим і валідованим, що робить його більш надійним.

- **Більша зручність для розробників**: Розробники можуть працювати над окремими модулями, що спрощує розподіл завдань та спільну роботу над проектом.

- **Легка масштабованість**: Модульна структура дозволяє легко масштабувати додаток, додавати нові сервери або ресурси для обробки більшої кількості користувачів.

- **Вищий рівень безпеки**: Модульна структура може допомогти уникнути витоку даних та інших безпекових проблем, оскільки кожен модуль може бути обмеженим у своїй функціональності та доступі.

- **Зручна інтеграція сторонніх сервісів**: Модульний додаток може легко інтегруватися зі сторонніми сервісами та API, що розширює його функціональність.

Загалом, модульна структура додатку сприяє його більш ефективному розвитку, обслуговуванню і масштабуванню, роблячи розробку і підтримку менш складними та ефективними.

Для більшого розуміння системи продемонструємо її загальний вигляд на рис. 2.1

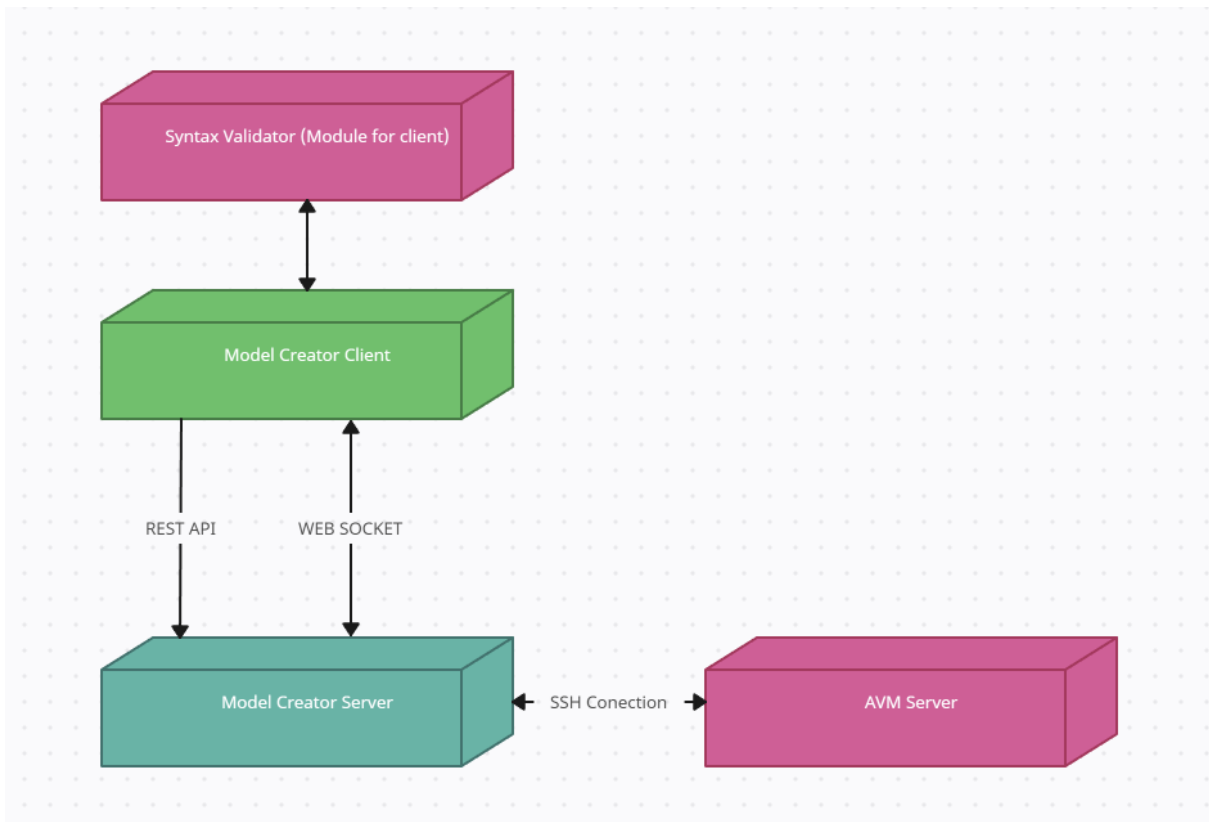

Рис 2.1 - Схема системи

На рисунку 2.1 видно, що основними об'єктами є розгалужені модулі, такі як Model Creator Client та Model Creator Server. Ці модулі взаємодіють між собою за допомогою інтерфейсу REST API та протоколу WebSocket. Ця взаємодія дозволяє їм обмінюватися важливою інформацією та виконувати різні операції.

Проте наш основний інтерес спрямований на модуль синтаксичного валідатора, який реалізований як додатковий модуль на стороні клієнта. Ця інформація є важливою для нас, оскільки вже існують раніше розроблені методи для ініціації валідації протоколів, поведінки, агентів та інших об'єктів. Отже, модуль валідації формул повинен працювати в тандемі з модулем синтаксичної валідації, щоб своєчасно та коректно надавати інформацію щодо помилок, пов'язаних з формулами.

Більш детальна взаємодія цих модулів показана на рисунку 2.2.

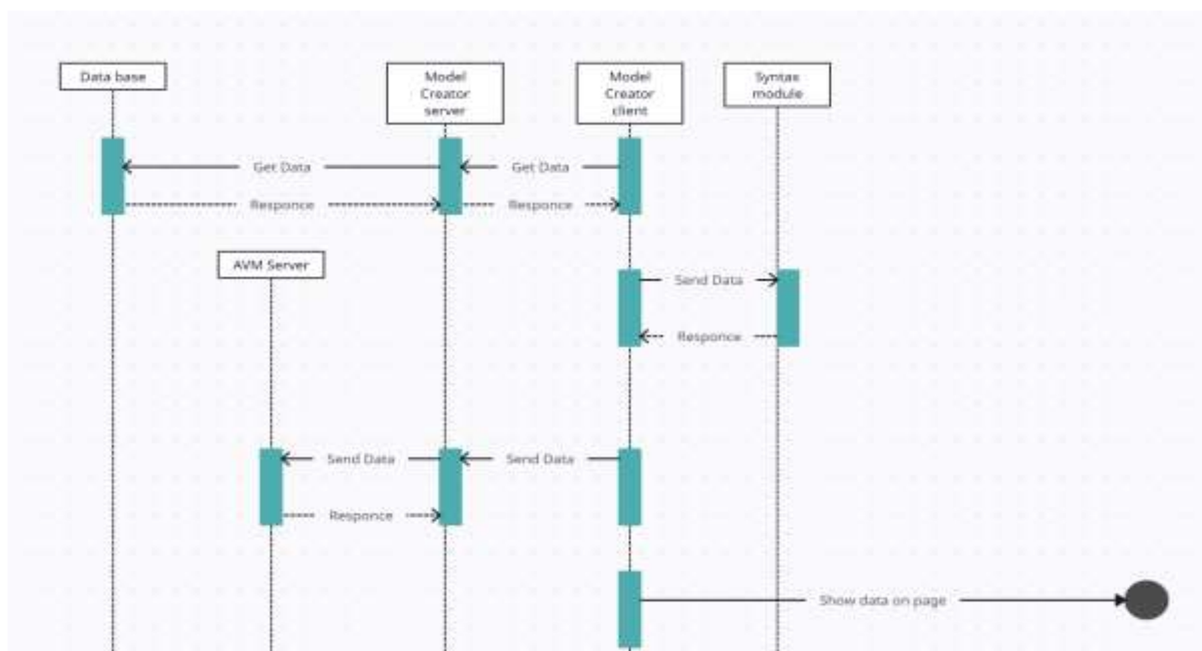

Рис 2.2 - Діаграма послідовності

2.1 Параметризований базовий протокол

Моделювання вставки постійно розвивається, і воно вносить нові стандарти та можливості щодо написання моделей. У попередніх версіях, користувачам доводилося створювати протоколи для кожного окремого агента (див. лістинг 2.1.1). Це означало, що вони повинні були писати ідентичні протоколи для різних агентів, а також керувати дублюванням коду. Необхідність вручну створювати і підтримувати такі протоколи призводила до значних втрат часу та ресурсів.

```
UNLOCKING = (
   (unlockTeam + !unlockTeam);
   (unlockInvestor + !unlockInvestor)
)
```

лістинг 2.1 - Протоколи для агентів

З появою більш сучасної версії стало можливим розробляти параметризований базовий протокол, який би був підходящим для кількох агентів в залежності від параметрів (див. лістинг 2.1.2).

```
UNLOCKING = (
   (unlock(team) + !unlock(team));
   (unlock(investor) + !unlock(investor))
)
```

лістинг 2.2 - Параметризовані протоколи

Проблема відсутності параметризованого базового протоколу була дуже помітною, оскільки вона спонукала користувача писати ідентичні протоколи для різних агентів, що в свою чергу призводило до значного витрат часу та створення однакового коду. За допомогою параметризованого базового протоколу ця проблема була вирішена. Тепер користувачі можуть створювати більш універсальні протоколи, які можуть налаштовуватися за допомогою параметрів відповідно до потреб кожного агента. Це значно спростило процес розробки та зменшило витрати на створення та управління кодом. Новий підхід дозволяє забезпечити більшу ефективність та рефакторинг коду, а також покращену повторне використання протоколів для різних сценаріїв. Таким чином, впровадження параметризованого базового

протоколу полегшило життя користувачів і покращило їхню продуктивність.

2.2 Валідація в Model Creator

Наше середовище розробки (Model Creator) виявилося нездатним підтримувати параметризовані базові протоколи, а точніше, валідатор не був призначений для обробки такого типу протоколів. Існуючий валідатор міг провести лише синтаксичну валідацію. Ця ситуація спонукала нас до розгляду можливості впровадження підтримки цих протоколів в нашому валідаторі.

2.2.3 Дерево вузлів.

Дерево вузлів (або "дерево рішень") - це структура даних і графічний спосіб подання інформації, яка використовується в областях, де потрібно приймати послідовні рішення або класифікувати дані в ієрархічному порядку. Дерева вузлів дуже поширені в різних галузях, таких як машинне навчання, штучний інтелект, аналіз даних, бізнес-процеси, біоінформатика та інші.

Основними складовими дерева вузлів є наступні поняття:

- **Корінь (Root)**: Початковий вузол, з якого розгалужуються всі інші вузли. Він знаходиться у вершині дерева і є його вихідним пунктом.

- **Вузли (Nodes)**: Вузли представляють рішення або події, які можуть відбуватися в системі. Кожен вузол може мати декілька наступників, які відображають можливі шляхи розвитку подій.

- **Гілки (Edges)**: Гілки з'єднують вузли між собою і показують послідовність вирішення рішень або переходів від одного вузла до іншого.

- **Листки (Leaves)**: Листки дерева - це кінцеві вузли, які не мають наступників. Вони представляють кінцеві результати або кінцеві стани системи.

- **Глибина (Depth)**: Глибина дерева вузлів вказує на кількість рівнів в дереві, яка відображається від кореня до найглибшого листка.

- **Поділ (Split)**: В процесі побудови дерева вузлів на кожному рівні приймається рішення про поділ даних на дві частини, і це поділ відбувається на основі конкретного критерію або параметра.

- **Класифікація і прогнозування**: Дерева вузлів часто використовуються для класифікації даних і прогнозування результатів на основі рішень, які вони приймають на кожному рівні.

Пояснимо, як працює дерево вузлів в нашій системі. Сам валідатор є окремим модулем, який отримує вхідні дані, а саме, текст, який потрібно перевірити, та синтаксичні правила. Цей модуль будує дерево вузлів, розділяючи кожен елемент тексту на окремі вузли. Кожен вузол має батьківський вузол і дерево дочірніх вузлів. Структура такого дерева дозволяє нам аналізувати кожний вузол

окремо і виявляти можливі синтаксичні помилки чи невідповідності в написаному коді. Цей підхід робить процес валідації швидшим і більш точним.

Для наочності розглянемо фрагмент модельного коду (див. лістинг 2.3).

```
price = x1 + x2;
```

лістинг 2.3 - Приклад операції

На першому етапі валідатор розбиває цей вузол на дерево вузлів. Нижче наведено приклад структури цього дерева (див. лістинг 2.4).

```
Node tree:
  token: ";"
  position: 15
  parent: -
  children:
        0:
              token: "="
              position: 6
                    parent: ";"
                    children:
                          0:
                                token: "price"
                                position: 0
                                parent: "="
                          1:
                                token: "+"
                                position: 11
```

parent: "="

children:

0:

token: "x1"

position: 8

parent: "+"

1:

token: "x2"

position: 13

parent: "+"

лістинг 2.4 - Дерево вузлів

Як бачимо, дерево, показане вище, надає виключну інформацію про положення кожного елемента переданого коду. Розподіл виконується відповідно до синтаксичного правила, яке було передано валідатору.

2.2.4 Синтаксичні правила.

Як було зазначено раніше, синтаксичні правила є необхідним елементом для нашого модуля валідації. Синтаксичне правило представляє собою набір норм і вимог, які описують структуру кожного компонента в нашому модельному коді. Надамо декілька прикладів.

Кожен ідентифікатор може складатися як із літер, так із цифр. Щоб визначити ідентифікатор як окремий структурний елемент, його необхідно відзначити в наших правилах(див. лістинг 2.5).

```
const IDENTITY:IRule = {
  name: 'Identity',
  regexp: /^([A-Za-z_]+)([0-9A-Za-z_]+)?$/
};
```

лістинг 2.5 - Синтаксичне правило для ідентифікатора

Як ми можемо побачити в цьому правилі(див. лістинг 2.5), за допомогою регулярного виразу вказано, що ідентифікатор може складатися з літер та цифр. Ідентифікатор - це один із трьох основних елементів, що позначає ім’я змінних, атрибутів, протоколів то що. Для чисел існує окремий ідентифікатор, описаний як "NUMBER", а для рядків - ідентифікатор "STR".

Давайте звернемо ближчу увагу на синтаксичне правило, яке базується на операції додавання, щоб краще зрозуміти принцип опису синтаксичних правил для операцій. Для прикладу ми розглядаємо найпростіший варіант операції додавання, тобто коли додаються два числа або змінна і число(див. лістинг 2.6).

```
const ADD:IRule = {
   name: 'add',
   patterns: [
     [{sequence: [IDENTITY, '+', NUMBER], keyElement: 1}],
     [{sequence: [NUMBER, '+', IDENTITY], keyElement: 1}],
     [{sequence: [NUMBER, '+', NUMBER], keyElement: 1}],
    [{sequence: [IDENTITY, '+', IDENTITY], keyElement: 1}],
   ]
  };
```

лістинг 2.6 - Синтаксичне правило для оператора додавання

Як видно в правилі(див. лістинг 2.6) для операції додавання, ми описали можливі варіації цієї операції. Зверніть увагу, що для прикладу ми взяли найпростіший варіант операції додавання, для більш складніших операцій потрібно описувати більш складні правила, щоб перекрити усі можливі варіанти.

2.2.3 Типізація.

Оскільки ідентифікатори можуть виступати як у ролі протоколів, агентів чи змінних, то для кожного з цих блоків створюється окреме дерево вузлів. Щоб перевірити можливість типізації елементів в межах певної поведінки чи протоколу, ми виконуємо наступні дії. Ми починаємо обхід отриманого дерева вузлів, а під час знаходження ідентифікатора ми надсилаємо запит до дерев протоколів, агентів, атрибутів чи типів, в залежності від місцезнаходження ідентифікатора в коді. Після виявлення ідентифікатора ми проводимо перевірку правильності його використання.

Давайте розглянемо це на прикладі: у нас є тип агента з ім'ям User, який має такий вигляд(див. лістинг 2.7):

User Agent Type:
rate: int
id: int

лістинг 2.7 - Тип агенту

Як ми можемо бачити(див. лістинг 2.7), у User існують два атрибути: id і rate. Допустимо, що у нас є агент користувача, який

належить типу User. Якщо ми спробуємо звернутися до неіснуючого атрибута, наприклад, user.amount, то в цьому випадку ми одразу отримаємо помилку, яка чітко та недвозначно вказує на спробу використання неіснуючого атрибута (див. рис. 2.3).

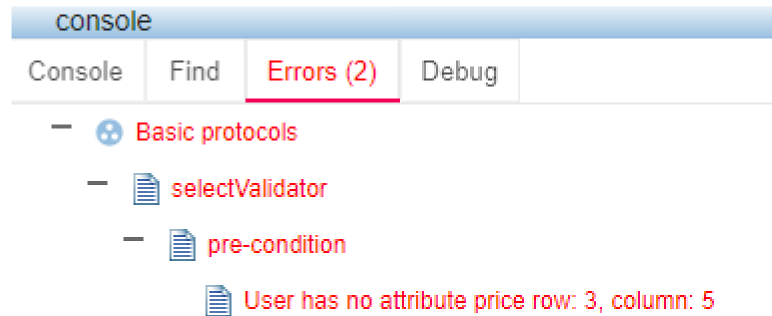

Рис 2.3 - Відображення помилки при спробі викликати неіснуючий атрибут

2.3 Валідація параметризованих протоколів

Одним із першочергових завдань було внести зміни до синтаксичних правил, а саме, додати до правил поведінки правила для протоколів з параметрами та інтегрувати їх у ці правила. Це необхідно для того, щоб валідатор міг побудувати правильну структуру дерева вузлів.

В першу чергу потрібно створити правило для аргументів протоколу(див. лістинг 2.8):

```
const BEH_ARGS:IRule = {
  name: 'behArgs',
  patterns: [
    [{sequence: [IDENTITY]}, {sequence: [',', IDENTITY], optional: true, repeatable: true}],
    [{sequence: [NUMBER]}, {sequence: [',', NUMBER], optional: true, repeatable: true}],
  ]
};
```

лістинг 2.8 - Правило для аргументів протоколу

Як можна побачити з цього переліку, правило для аргументів визначає, що саме може бути аргументами. У нашому випадку, це ідентифікатори, числа та їх списки.

Наступним етапом є модифікація правила запису протоколу в моделі поведінки(див. лістинг 2.9):

```
const BEH_PROTOCOL:IRule = {
  name: 'protocol',
  patterns: [
    [{sequence: [IDENTITY, '(', BEH_ARGS, ')'], keyElement: 0}],
    [{sequence: [IDENTITY, '(', ')'], keyElement: 0}],
    [{sequence: [IDENTITY]}]
  ]
};
```

лістинг 2.9 - Правило для параметризованного протокола

Ви можете побачити(див. лістинг 2.9), що ми додали правило, що описує, що протокол може мати аргументи. Як ви можете побачити, раніше ці протоколи можна було писати без аргументів і навіть без круглих дужок.

Отже, зараз валідатор правильно будує дерево вузлів, але це лише перший етап, потрібно додати ще обробник до цього дерева вузлів. Зрозуміло, що в аргументах протоколу можна вказувати не лише ідентифікатор чи число, а також вираз, наприклад, результат додавання двох елементів(див. лістинг 2.10):

Protocol(element1 + element2)

лістинг 2.10 - Параметризований протокол з виразом в аргументах

Для пошуку ситуації, яка вважається помилкою "Невідомий токен", валідатор перевіряє дочірні вузли протоколів або поведінки, щоб визначити, чи належать вони до атрибутів, аргументів або параметрів. Тому ми отримаємо цю помилку, якщо напишемо вираз у параметрах протоколу, оскільки знаки операцій не будуть відповідати більш як одному типу пошуку. Рішенням цього є дуже проста модифікація перевірки, а саме, додавання опції, яка дозволить використовувати ці знаки операцій. Тобто додати до правила аргументів протоколу наступний параметр(див. лістинг 2.11):

[{sequence: [EXPRESSION]}, {sequence: [',', EXPRESSION], optional: true, repeatable: true}],

лістинг 2.11 - Додатковий параметр для правила аргументів протоколу

Отже після налаштування синтаксичних правил, які допоможуть правильно будувати дерево вузлів з поведінки моделі, можемо починати розробку валідації параметризованих протоколів. Так як ми будуємо новий функіонал над вже існуючим модулем для синтаксично валідації то необхідно вбудовуватися в функцію яка займається валідацією післяумови та передумови протоколу. В нашому випадку такою функцію є фунція parseCondition(див. рис.2.3.1).

```
parseCondition(text:string, action:IAction, allowMultiple = false):ICompileResult {
  const attrs = this.getAttrsFromQuantifier(action);
  if (action.args) {
    action.args.forEach((a) => attrs.set(a, ANY_TYPE));
  }

  let parsedBeh: IAST | null = null;

  if (action.args) {
    if (action.args.length > 0) {
      parsedBeh = astConverter(BEH_WITH_RS_GRAMMAR).convert(this.model.behavior.code);
    }
  }

  if (!allowMultiple) {
    return parseExpression(text, this.model, attrs, LOGICAL_GRAMMAR, undefined, action, parsedBeh);
  }

  const splitted = new CommentManager(text).splitted();
  const blocks:string[] = [];
  splitted.forEach((cp) => {
    if (cp.isComment) {
      blocks.push(cp.value);
    } else {
      blocks.push(...(cp.value.split(';').filter((s) => s.length)));
    }
  });
  if (blocks.length) {
    const exprText = blocks.shift() || '';
    const res = parseExpression(exprText, this.model, attrs, POST_CONDITION_GRAMMAR, undefined, action, parsedBeh);
    const quantifiable:ICompileResult = {annotations: res.annotations, markers: res.markers};
    const prevTexts = [exprText];
    while (blocks.length) {
      const nextText = blocks.shift() || '';
      const nestExpr = parseExpression(nextText, this.model, attrs, POST_CONDITION_GRAMMAR, undefined, action, parsedBeh);
      const nest:ICompileResult = {annotations: nestExpr.annotations, markers: nestExpr.markers};
      if (!quantifiable.annotations) {
        quantifiable.annotations = [];
      }
      if (!quantifiable.markers) {
        quantifiable.markers = [];
      }
      quantifiable.annotations.push(...addAnnotationsOffset(prevTexts, nest.annotations || []));
      quantifiable.markers.push(...addMarkersOffset(prevTexts, nest.markers || []));
      prevTexts.push(nextText);
    }
    return quantifiable;
  }
  return {annotations: [], markers: []};
}
```

Рис 2.3.1 - Функція parseCondition

Функція *parseCondition*, згадана у попередньому тексті (див. рис.2.3.1), грає ключову роль в обробці протоколів. Вона відповідає за розбиття протоколу на окремі частини, а саме квантор, передумову та післяумову. Цей розподіл важливий для подальшого аналізу і валідації протоколу.

Процес роботи функції parseCondition включає наступні етапи:

- **Визначення типу протоколу**: Першочергово функція перевіряє, чи оброблюваний протокол є параметризованим. Це важливо, оскільки ми будемо розглядати виклик лише для параметризованих протоколів. Якщо протокол є параметризованим, то він підлягає подальшій обробці; в іншому випадку функція може пропустити його.

- **Виділення квантора, передумови та післяумови**: Функція розбиває вміст протоколу на окремі частини, а саме квантор, передумову і післяумову. Квантор вказує на кількість ітерацій в протоколі, передумова - умови, які повинні бути виконані перед виконанням протоколу, і післяумова - очікувані результати після виконання протоколу.

- **Передача частин для аналізу**: Після розподілу частин протоколу, функція *parseCondition* передає кожну частину окремо для подальшого аналізу функції *parseExpression*. Це допомагає виконати більш глибокий аналіз та валідацію кожної частини протоколу.

Процес перевірки протоколу в цій функції не дуже складний так як ми маємо параметри *action* типу *IAction*. Давайте більш детальніше розглянемо тип *IAction* (див. рис. 2.3.2).

```
export interface IAction extends IValidated{
    name:string;
    args?:string[];
    entities?:IAgent[];
    quantifier?:IQuantifiable;
    preCondition:IQuantifiable;
    process?: IProcess[];
    postCondition:IQuantifiable;
    msc:IMSCConfig;
    getEntitiesNames():string[];
}
```

Рис 2.3.2 - Тип даниї IAction

Як показано на рисунку 2.3.2, тип даних *IAction* має різні поля, включаючи *name*, *args*, *entities*, *quantifier*, *preCondition*, *process*, *postCondition*, та *msc*. Насамперед, нас цікавить поле *args*, оскільки це саме поле містить аргументи протоколу і надає нам можливість визначити, чи є цей протокол параметризованим.

Поле *args* в *IAction* є масивом (або списком) аргументів, які передаються в протоколі під час його виконання. Ці аргументи можуть бути різного типу, такі як числа, рядки, об'єкти, інші типи даних чи посилання на інші об'єкти. Аналіз поля args дозволяє нам визначити, чи є протокол параметризованим, оскільки в параметризованих протоколах аргументи зазвичай представлені змінними чи параметрами, які можуть варіюватися в різних ітераціях чи викликах протоколу.

Якщо масив *args* містить змінні, які можуть приймати різні значення в залежності від контексту або параметрів, то це може свідчити про параметризацію протоколу. В іншому випадку, якщо args містить сталий набір значень, протокол вважатиметься непараметризованим.

Отже, аналіз поля *args* допомагає визначити, чи є протокол параметризованим, і відіграє важливу роль у роботі з протоколами та визначенні їх характеристик.

```
if (action.args) {
  if (action.args.length > 0) {
    parsedBeh = astConverter(BEH_WITH_RS_GRAMMAR).convert(this.model.behavior.code);
  }
}
```

Рис 2.3.3 - Тип даниї IAction

Можемо побачити (див. рис.2.3.3), що спочатку перевіряється існування *args* в *action* , це робиться для того щоб не потрапити на виклик undefined аргумента тому що args може не існувати (див. рис.2.3.2). Коли існування *args* підтверджено ми проводимо перевірку що довжина масиву більше нуля, а це в свою чергу говорить нам про те що цей протокол є параметризованим, і тіки в цьому випадку ми викликаємо функцію *astConverter* (див. рис. 2.3.3), який проводить побудову дерева вузлів з переданої частини коду, тобто поведінки моделі. Побудова дерева виконується за вказаною граматикою, тобто списком синтаксичних правил, в нашому випадку це *BEH_WITH_RS_GRAMAR* , що відповідає за граматику поведінки моделі. В результаті ми отримуємо дерево вузлів поведінки, яке передається до функції *parseExpression*, і наявність якого підтверджує

параметризованість переданого протоколу і ініціює подальшу перевірку.

Після виявлення, що протокол є параметризованим, нам потрібно визначити, які аргументи були передані цьому протоколу в поведінці, це необхідно для подальшої перевірки. Це вимагає отримання дерева вузлів моделі поведінки та подальшого обходу до нашого ідентифікатора, який вже був визнаний як протокол. Для обробки дерева вузлів ми використовуємо рекурсивну функцію *getActArg*(див. рис.2.3.4).

```
const getActArg = (behNode: IASTNode, actionName: string): IASTNode[] => {
  let argArr: IASTNode[] = [];
  if (behNode.parent && String(behNode.parent.token.value) === actionName) {
    argArr.push(behNode);
  }
  if (behNode.children) {
    behNode.children.forEach((child) => {
      const res = getActArg(child, actionName);
      argArr = argArr.concat(res);
    });
  }
  return argArr;
}
```

Рис 2.3.4 - функція *getActArg*

На рисунку 2.3.4 видно, що функція getActArg виконує одну просту, але надзвичайно важливу задачу. Вона відповідає за пошук аргументів, переданих протоколу в поведінці. Основна мета цієї функції - визначити, чи існує "батьківський" вузол у поточному контексті та, якщо так, перевірити, чи співпадає ім'я цього "батька" з ім'ям шуканого протоколу. Якщо ця умова виконується, то вузол додається до результату. У випадку невідповідності умови починається обхід дочірніх вузлів з подальшим рекурсивним

викликом функції getActArg для кожного з дочірніх вузлів. Крім того, функція отримує та зберігає всі вузли, які є переданими параметрами для даного протоколу, у спеціальному масиві, щоб уникнути повторного обходу дерева. Це дозволяє ефективно виконувати пошук та аналіз структури даних, яка розглядається, і відіграває ключову роль в багатьох аспектах функціонування системи. Давайте розглянемо приклад. Нехай початковий виклик протоколу виглядає так, як показано в лістингу 2.12.

Protocol(element1 , element2 + element3)

лістинг 2.12 - Приклад виклику протоколу

Отже, після процедури отримання вузлів аргументів (виконання функції *getActArg* , див. рис. 2.3.4), результуючий масив буде містити такі вузли, наприклад(див. лістинг 2.13):

Node:

token: ",",

parent: "(",

children:

0:

token: "element1",

parent: ","

1:

token: "+",

parent: ",",

children:

0:

token: "element2",

parent: "+"

1:

token: "element3",

parent: "+"

лістинг 2.13 - “Сирий” результуючий масив

На лістингу можна побачити, що результуючий масив містить вузол, в якому присутні два дочірні вузли. Це важливий момент, оскільки він вказує на наявність більш складної структури вхідних даних.

Перший вузол містить токен "*element1*", який вже є параметром інтересу. Це означає, що в цьому вузлі міститься необхідна інформація для нашого аналізу.

У другому вузлі ми бачимо токен "+", що свідчить про те, що другий вузол посилає нас на вираз. Цей вузол є операційним символом, і його дочірні вузли є елементами цього виразу. В цьому випадку, дочірні вузли "*element2*" та "*element3*" вказують на токени, які також можуть бути потенційними аргументами або параметрами.

Ця взаємодія токенів та вузлів у "сирому" масиві показує, як різні типи даних та операції можуть взаємодіяти та створювати більш складні структури для обробки. Обробка таких структур вимагає глибокого аналізу та відокремлення корисної інформації для подальшого використання у системі.

Після обробки функцією getActArg ми отримуємо "сирі" масиви, які, в принципі, можуть бути менш зручними для подальшого аналізу. Ці масиви можуть містити декілька вузлів, і не обов'язково всі аргументи будуть розташовані в одному вузлі. Така структура не відповідає нашим потребам для подальшого аналізу, оскільки нам необхідні лише ідентифікатори аргументів. Тому ці "сирі" масиви потребують додаткової обробки.

Спочатку ми перевіряємо кожний токен в кожному вузлі, що міститься у "сирому" масиві аргументів. Ми перевіряємо, чи цей токен є ідентифікатором або числом. Якщо це так, то ми зберігаємо цей елемент в результуючому масиві аргументів, який буде використовуватися в подальших обчисленнях або операціях. Однак, якщо токен є спеціальним символом операцій або іншого типу, то ми розпочинаємо рекурсивний аналіз дочірніх вузлів цього символу.

Цей процес допомагає відфільтрувати і відокремити корисну інформацію від зайвої та підготувати дані для подальшої обробки. В результаті отримуємо структуровані дані, які дозволяють здійснювати більш точний та ефективний аналіз, спрощуючи подальші операції з цими даними.

Для ефективної обробки "сирого" масива була створена функція makeActArg (див. рисунок 2.3.5). Ця функція відповідає за обхід переданої частини дерева вузлів і перевірку кожного токену в цьому контексті. Перевірка виконується за допомогою регулярного виразу, який визначає, чи цей токен є ідентифікатором чи числом. Якщо токен відповідає цим критеріям, він додається до результуючого масиву для подальшого використання.

У випадку, якщо перевірка не пройшла, функція розпочинає обхід дочірніх вузлів (якщо вони існують). Цей обхід дочірніх вузлів виконується за допомогою рекурсивного виклику самої функції makeActArg (див. рис. 2.3.5), при цьому передається дочірній вузол для подальшого аналізу. Після обходу дочірніх вузлів до результату цього виклику додатково додається батьківський токен, який розташовує між двома дочірніми токенами (див. лістинг 2.14.). Це важливий крок, який допомагає побудувати структуру даних, відображаючи її відносини та ієрархію.

Функція makeActArg відіграє вирішальну роль у перетворенні неструктурованого "сирого" масиву в структурований формат, який легше аналізувати та обробляти в подальших операціях. Такий підхід допомагає зробити дані більш доступними та корисними для системи.

```
const makeActArg = (nodeList: IASTNode[]): any[] => {
  const result: any[] = [];
  nodeList.forEach((element) => {
    if (/^([A-Za-z_]+)([0-9A-Za-z_]+)?$/.test(element.token.value)) {
      result.push(element.token.value);
    } else {
      if (element.children) {
        const res = makeActArg(element.children);
        res.splice(res.length / 2, 0, element.token.value);
        result.push(res);
      } else {
        result.push(element.token.value);
      }

    }
  });
  return result;
}
```

Рис 2.3.5 - функція *makeActArg*

Ця процедура є важливою ланкою у відокремленні та відібранні інформації, що дозволяє нам отримати більш розгорнутий та інформативний масив аргументів, в якому лише зберігаються ідентифікатори, які мають значення для подальшої перевірки та аналізу.

Внаслідок цієї процедури, ми одержуємо масив аргументів, який має наступний вигляд, схожий на той, який показаний у лістингу 2.14. Він містить лише ідентифікатори аргументів, які мають ключове значення для нашого дослідження та взаємодії в системі. Ця обробка даних допомагає скоротити обсяг та спростити подальший аналіз, забезпечуючи необхідну інформацію для ефективної роботи з даними та оптимізації процесів системи.

[“element1”, “element2”, “+”,” element3”]

лістинг 2.14 - Послідовний масив ідентифікаторів та операторів

Зауважте, що ми залишили знаки операцій у виразах. Це було зроблено з певною метою, а саме - для можливості правильного розділення ідентифікаторів від виразів, які можуть бути аргументами. Це розділення є важливим кроком, оскільки воно дозволяє здійснити наступний етап - перевірку кількості аргументів, переданих у протокол.

Залишаючи знаки операцій у виразах, ми створюємо чітку розмежування між окремими компонентами виразу, такими як ідентифікатори і операції. Це дозволяє нам легко виділити ідентифікатори, які є потенційними аргументами протоколу, і подальше їх аналізувати.

Такий підхід полегшує завдання з перевірки кількості аргументів, переданих у протокол, оскільки ми можемо використовувати знаки операцій як роздільники для ефективного підрахунку та ідентифікації окремих аргументів. Ця система допомагає уникнути помилок та спрощує обробку вхідних даних.

Для цього ми отримуємо інформацію про сам протокол, використовуючи його ідентифікатор. Ця інформація включає в себе список параметрів, які приймає протокол. Після отримання цієї інформації та знаючи необхідну кількість аргументів, яку повинен приймати протокол, ми повторно викликаємо функцію *validateNode*(див. рис. 2.3.6), яка є частиною синтаксичного парсера, ця функція допоможе порівняти отриманні дані зі списком аргументів, які ми передали, і враховуємо можливість виразів серед аргументів. Тепер ми можемо виявити одну з перших помилок - чи була передана необхідна кількість аргументів (див. Рис. 2.4, Рис. 2.5).

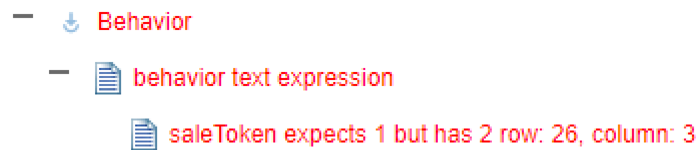

Рис 2.4 - Повідомлення про помилку під час введення більшої кількості аргументів, ніж очікувалося

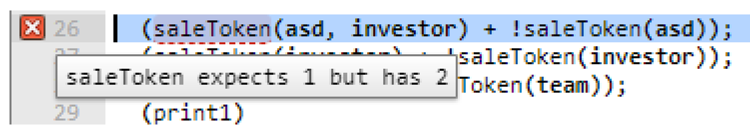

Рис 2.5 - Маркер і підказка, що пояснює, де сталася помилка

Рис 2.3.6 - Пошук агенту і виклик *validateNode*

Як можна побачити на Рисунку 2.3.6, для перевірки чи була передана необхідна кількість аргументів (див. Рисунок 2.4, Рисунок 2.5), ми виконуємо наступну послідовність дій. По-перше, починаємо обхід списку аргументів, які були вказані в протоколі. Після цього ми перевіряємо, чи ці аргументи відповідають поточному вузлу, який обробляється. Якщо знайдено відповідність, ми переходимо до наступного етапу.

Далі ми починаємо обхід масиву з аргументами, які ми отримали з поведінки. Наша мета - знайти агента в моделі, який відповідає аргументу, отриманому з поведінки. Це необхідно для подальшого аналізу та перевірки.

Якщо ми успішно знаходимо такого агента, ми викликаємо функцію *validateNode* (див. Рисунок 2.3.6). Ця функція починає рекурсивний процес перевірки нашого протоколу, підставляючи знайденого агента на місце визначеного аргумента. Після закінчення аналізу ця функція повертає масив даних, в якому можуть бути включені помилки та маркери помилок, якщо такі виявлені.

Цей комплексний процес дозволяє системі впевнитися, що передані аргументи відповідають вимогам протоколу та відповідним агентам в моделі. В разі виявлення невідповідностей система може надати інформацію про помилки, що допомагає виправити їх та забезпечити коректну роботу системи.

```
if (operator.input.length) {
  errors.push(createAnnotation(
    str,
    node.token.position,
    `${node.token.value} expects ${operator.input.length} arguments`
  ));
  markers.push(createMarker(str, node.token.position, node.token.value));
}
```

Рис 2.3.7 - Перевірка на кількість аргументів в *validateNode*

На Рисунку 2.3.7 наведено фрагмент коду функції *validateNode*, який ілюструє, як ми генеруємо та обробляємо помилки. У цьому фрагменті коду відображено процес генерації помилки, що включає в себе структуру для зберігання тексту помилки та її позиції.

Під час генерації помилки, ми створюємо об'єкт, який містить інформацію про помилку, таку як текст помилки та позиція, де вона виникла. Ця інформація може бути використана для створення анотації, яка буде додана до масиву помилок. Анотація містить детальний опис помилки, її характеристики та контекст, що полегшує подальший аналіз помилок.

Додатково до створення анотації, функція генерує маркер, який допомагає знаходити та відображати помилку в коді. Маркери можуть включати в себе інформацію про те, як і де помилка пов'язана з вихідним кодом, що допомагає розробникам швидко локалізувати та виправляти проблему(див. рис. 2.5).

Такий підхід дозволяє системі ефективно виявляти, обробляти та документувати помилки, що виникають під час аналізу протоколів та взаємодії в системі, сприяючи подальшій розробці та підтримці системи.

Цей процес має велике значення для визначення коректності структури коду та уникнення помилок, пов'язаних з неправильними типами даних, переданими в аргументах протоколу. Попереднє виявлення невірних аргументів на етапі проектування моделі є надзвичайно важливим і цілком обґрунтованим. Від цього етапу залежить ефективність та надійність функціонування системи.

Виявлення неправильних аргументів на етапі проектування є менш затратним і менш проблематичним порівняно з ситуацією, коли такі помилки виявляються під час запуску моделі з невірними аргументами. У такому випадку, помилки від AVM (Алгебраїчна

віртуальна машина) можуть бути меншою проблемою, але наслідки невірних аргументів можуть бути дуже серйозними.

Неправильні аргументи можуть призвести до невірного моделювання в системі, що, в свою чергу, може вплинути на правильність результатів, які надаються системою. Це може призвести до недорозумінь, помилок у прийнятті рішень, або навіть до критичних помилок, які можуть завдати шкоди користувачам та системі загалом.

Таким чином, попереднє визначення та перевірка аргументів протоколу мають вирішальне значення для забезпечення стабільності та надійності системи, а також для попередження потенційних проблем, які можуть виникнути в процесі її роботи.

Зараз, на цьому етапі, ми розполагаємо всією необхідною інформацією для подальшої важливої перевірки. Іншими словами, ми вже визначили аргументи протоколу в моделі поведінки та отримали інформацію про цей протокол у його оголошенні. Проте тепер настав час для більш докладної перевірки.

Наш наступний крок - перевірити, чи відповідають типи аргументів, які ми передаємо, типам аргументів, які очікує протокол. Ця перевірка стане остаточним етапом у визначенні сумісності аргументів між моделлю поведінки та протоколом. Вона дозволить впевнитися, що дані, які передаються, відповідають очікуванням протоколу і правильно взаємодіють з ним.

Для подальшої перевірки розпочинаємо звертанням до дерева Агентів, щоб отримати важливу інформацію про наш аргумент. Варто зауважити, що ця процедура вже була виконана раніше під час пошуку агента для попередньої помилки (див. Рисунок 2.3.6). У процесі отримання цієї інформації, ми також дізнаємося тип цього агента, який надає нам важливу інформацію про природу даного аргумента.

Маючи цей тип агента та знаючи його позицію у дереві, ми можемо провести порівняння з типом аргумента, який очікується на цій конкретній позиції відповідно до оголошення протоколу. Ця порівняльна процедура дозволяє нам переконатися, що аргумент, який ми передаємо, відповідає вимогам протоколу, і що вони сумісні за типами.

Цей етап перевірки гарантує, що дані, які передаються між агентами та протоколами, відповідають всім необхідним обмеженням та вимогам, встановленим в оголошенні протоколу. Ця процедура важлива для запобігання можливим конфліктам, помилкам та неправильній обробці даних, що можуть виникнути під час взаємодії в системі.

Зрівнявши ці два типи, ми отримуємо результат, який може включати в себе можливість виявлення помилки. Якщо типи не збігаються, це може свідчити про невідповідність між тим, що було передано, і тим, що очікується в рамках протоколу. Ця перевірка є ключовою для забезпечення правильності та безпеки взаємодії між агентами та протоколами, оскільки допомагає запобігти потенційним конфліктом та помилкам в системі.

Виявлення невідповідності типів дозволяє на ранніх етапах ідентифікувати можливі помилки та проблеми у взаємодії агентів та протоколів. Ця перевірка виконує важливу функцію у впровадженні високої якості та надійності в систему. Крім того, вона виступає як захисний механізм, що допомагає уникнути невідповідностей, конфліктів та помилок, що можуть виникнути в подальшому експлуатації системи.

Отже, ця перевірка грає ключову роль у забезпеченні правильності та надійності системи, а також в сприянні вчасному виявленню та вирішенню можливих проблем, що можуть виникнути в процесі її роботи.

Як показано на Рисунку 2.3.6, під час виклику функції *validateNode*, яка оперує переданими даними, ми запускаємо подальшу обробку нашого протоколу з заданим аргументом. В цей момент функція не виконує блок пошуку параметрів, оскільки ми вже надали дані [agent.name, index, agent.type.name], які свідчать функції про відсутність необхідності в подальшому пошуку протоколу. Таким чином, функція відразу переходить до перевірки сумісності типів аргументу з типами, очікуваними в рамках протоколу.

У випадку, коли виявляється несумісність типів, функція генерує помилку та маркер (див. Рисунок 2.3.8). Генерація помилки є важливим кроком, оскільки вона вказує на конфлікт між переданим аргументом і вимогами протоколу, дозволяючи вчасно виявити та вирішити цю проблему. Маркер, у свою чергу, допомагає локалізувати місце, де виникла помилка, і швидко знайти її у вихідному коді для подальшої корекції.

Отже, цей процес виконує важливу функцію в забезпеченні коректної взаємодії агентів та протоколів, а також у вчасному виявленні та усуненні можливих помилок в системі.

Рис 2.3.8 - Генерація помилок несумісності типів очікуємого та переданого аргумента

Як ми раніше зазначали, помилки та маркери, що генеруються в коді для виявлення несумісності типів між очікуваним та переданим аргументами, як показано на Рисунку 2.3.8, автоматично відображатимуться в інтерфейсі Model Creator (див. Рисунок 2.6, Рисунок 2.7) для користувача. Це важливий елемент забезпечення якості та коректності моделей, оскільки дозволяє користувачу негайно сприймати та реагувати на помилки, що виникають у процесі створення моделі.

Завдяки цьому механізму користувач може швидко і точно локалізувати місце, де виникла помилка, та вжити необхідні кроки для її усунення. Це зроблено з метою покращення ефективності та продуктивності роботи з моделями поведінки, забезпечуючи користувачу зручність та швидкість у виявленні та виправленні потенційних проблем.

Рис 2.6 - Повідомлення про помилку, пов’язане з типом переданого аргументу, і помилка про те, що саме не так із типом

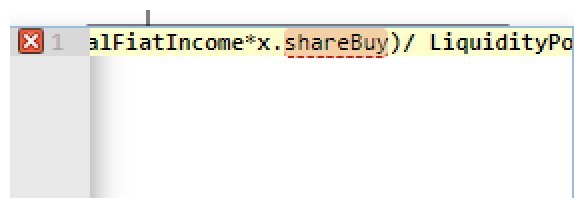

Рис 2.7 - Маркер і спливаюча підказка у вікні редагування умови протоколу з поясненням помилки

ВИСНОВКИ

У ході цієї роботи було проведено аналіз, розробку та інтеграцію важливого компонента для системи IMS. Важливі висновки та досягнення цієї роботи включають наступне:

- **Аналіз формул та їх структури**: Проведений детальний аналіз синтаксису та структури формул дозволив краще зрозуміти особливості роботи з ними. Цей етап дав змогу визначити, як вони будуть взаємодіяти в системі IMS та які можливі проблеми можуть виникнути.

- **Розробка валідатора формул**: Розроблений валідатор формул, який є ключовим компонентом для забезпечення правильності формул у системі IMS. Цей валідатор дозволяє виявляти та усувати помилки у формулах, підвищуючи якість та надійність обробки даних.

- **Реалізація з використанням JavaScript та TypeScript**: Валідатор був успішно реалізований за допомогою мов програмування JavaScript та TypeScript на базі операційної системи Ubuntu. Ця реалізація дозволяє ефективно використовувати валідатор у системі IMS.

- **Визначення та розділення аргументів**: Валідатор виконує важливу функцію визначення та розділення аргументів у формулах, забезпечуючи правильну передачу даних та уникнення помилок.

- **Перевірка правильності передачі аргументів та пошук помилок**: Валідатор перевіряє правильність передачі аргументів та виявляє помилки у коді поведінкових моделей. Це допомагає у виявленні можливих проблем та забезпечує коректну обробку даних.

- **Тестування та інтеграція в систему IMS**: Розроблений валідатор був успішно підданий тестуванню на реальних даних та інтегрований в систему IMS, а саме в середу розробки Model Creator. Це забезпечує його ефективне та надійне використання в реальних умовах.

У підсумку, ця робота має велике значення для забезпечення якості та надійності обробки формул у системі IMS, допомагаючи виявляти та усувати можливі помилки, що можуть виникнути при їх обробці. Розроблений валідатор є важливим інструментом для підтримки та покращення функціональності системи IMS.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Letichevsky, A. A., Letychevskyi, O. A., & Peschanenko, V. S. (2012). Insertion modeling system. In Perspectives of Systems Informatics: 8th International Andrei Ershov Memorial Conference, PSI 2011, Novosibirsk, Russia, June 27-July 1, 2011, Revised Selected Papers 8 (pp. 262-273). Springer Berlin Heidelberg.

2. Letichevsky, A. A., Kapitonova, J. V., Letichevsky Jr, A. A., Kotlyarov, V. P., Nikitchenko, N. S., Volkov, V. A., & Weigert, T. (2008). Insertion modeling in distributed system design.

3. Letichevsky, A., Letychevskyi, O., & Peschanenko, V. (2016). Insertion modeling and its applications. Computer Science Journal of Moldova, 72(3), 357-370.

4. Olexandr A. Letychevskyi, Vladimir S. Peschanenko: Applying Algebraic Virtual Machine to Cybersecurity Tasks. In: 2022 IEEE 9th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT), IEEE, Hammamet, Tunisia(2022)

5. Letychevskyi, O., Peschanenko, V., & Volkov, V. (2021, September). Algebraic virtual machine project. In International Conference on Information and Communication Technologies in Education, Research, and Industrial Applications (pp. 353-364). Cham: Springer International Publishing.

6. Letychevskyi, O., Peschanenko, V., & Volkov, V. (2021, September). Algebraic Virtual Machine and Its Applications. In International Conference on Information and Communication Technologies in Education, Research, and Industrial Applications (pp. 23-41). Cham: Springer International Publishing.

7. Bierman, G., Abadi, M., & Torgersen, M. (2014). Understanding typescript. In ECOOP 2014–Object-Oriented Programming: 28th European

Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28 (pp. 257-281). Springer Berlin Heidelberg.

8. Richards, G., Zappa Nardelli, F., & Vitek, J. (2015). Concrete types for TypeScript. In 29th European Conference on Object-Oriented Programming (ECOOP 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
9. Fenton, S., Fenton, & Spearing. (2014). Pro TypeScript. Apress.
10. Di Rienzo, J. A., Guzmán, A. W., & Casanoves, F. (2002). A multiple-comparisons method based on the distribution of the root node distance of a binary tree. Journal of agricultural, biological, and environmental statistics, 7, 129-142.
11. Satheesh, M., D'mello, B. J., & Krol, J. (2015). Web development with MongoDB and NodeJs. Packt Publishing Ltd.
12. Rimal, A. (2019). Developing a web application on NodeJS and MongoDB using ES6 and beyond.
13. Gackenheimer, C. (2015). Introduction to React. Apress.
14. Fedosejev, A. (2015). React. js essentials. Packt Publishing Ltd.
15. Boduch, A. (2017). React and react native. Packt Publishing Ltd.