**МАГІСТЕРСЬКА РОБОТА**

МР. ІПм - 49.00.00.000 ПЗ

**Група ІПм-23-2**

**Мацайло Назарій**

**2024**

**Івано-Франківський національний технічний університет нафти і газу**

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

**Мацайло Назарій Васильович**

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

# МАГІСТЕРСЬКА РОБОТА

**Методи доменно-специфікованих мов для супервайзингу генерованих аплікацій**

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Мацайло Н.В.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

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

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

**Допущено до захисту**

Завідувач кафедри

доц. Бандура В.В.

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

Нормоконтроль

доц. Вовк Р.Б.

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

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

Івано-Франківськ – 2024

**Івано-Франківський національний технічний університет нафти і газу**

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

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

# ЗАВДАННЯ

## НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

**Мацайлу Назарію Васильовичу**

(прізвище, ім’я, по-батькові)

**1. Тема магістерської роботи “ Методи доменно-специфікованих мов для супервайзингу генерованих аплікацій ”**

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

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

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

**3. Вихідні дані до проекту (роботи)** Теоретичні концепції та формальні моделі побудови та функціонування інформаційних та програмних технологій доменно-специфікованих мов

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

1. Дослідження предметної області застосування доменно-специфікованих мов
2. Моделі та методи використання доменно-специфікованих мов супервайзингу в робототехніці
3. Підхід застосування доменно-специфікованої мови до процесів супервайзингу
4. Імплементація методів та моделей доменно-специфікованих мов для супервайзингу аплікацій

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

1. Схема модельно-орієнтованої трансформації (рис. 1.1)
2. Методи трансформації M2M і M2T (рис. 1.2)
3. Інструмент JetBrains MPS (рис. 1.3)
4 Платформа Visual-ROS (рис. 1.4)
5. Приклад графа ROS з трьома вузлами (рис. 1.5)

**6. Консультанти розділів проекту (роботи)**

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

**7. Дата видачі завдання** 04 вересня 2024 р.

**Керівник** _____
(підпис)

**Завдання прийняв до виконання**_____
(підпис)

# КАЛЕНДАРНИЙ ПЛАН

| № п/п | Назви етапів магістерської роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Підбір і вивчення літератури по темі магістерської роботи | 15.09.2024 | виконано |
| 2 | Аналіз концепцій та алгоритмів предметної області | 29.09.2024 | виконано |
| 3 | Дослідження предметної області застосування доменно-специфікованих мов | 15.10.2024 | виконано |
| 4 | Моделі та методи використання доменно-специфікованих мов супервайзингу в робототехніці | 08.11.2024 | виконано |
| 5 | Підхід застосування доменно-специфікованої мови до процесів супервайзингу | 20.11.2024 | виконано |
| 6 | Імплементація методів та моделей доменно-специфікованих мов для супервайзингу аплікацій | 01.12.2024 | виконано |
| 7 | Затвердження пояснювальної записки роботи завідувачем кафедри | 15.12.2024 | виконано |

**Студент – магістр** _____
(підпис)

**Керівник роботи** _____
(підпис)

# АНОТАЦІЯ

**Магістерська робота:** 78 с., 28 рис., 2 табл., 50 джерел.

**Тема:** Методи доменно-специфікованих мов для супервайзингу генерованих аплікацій

**Об'єкт дослідження:** складні системи, що функціонують у динамічних середовищах і потребують супервайзингового управління.

**Мета роботи:** імплементувати підходи, методи і моделі використання доменно-специфікованих мов для супервайзингового управління робототехнічними системами.

**Предмет дослідження:** методи, моделі та технології використання доменно-специфікованих мов у процесах супервайзингового управління.

**Результати дослідження**

В роботі інтегровано теорію супервайзингового контролю в процеси розробки робототехнічних систем із використанням MDE та удосконалено методи імітаційного моделювання супервайзингових процесів у складних системах.

**Висновок**

Результати роботи свідчать про те, що використання доменно-специфікованих мов у поєднанні з модельно-керованою інженерією та теорією супервайзингового контролю є перспективним напрямом у розробці робототехнічних систем

**ДОМЕННО-СПЕЦИФІКОВАНІ МОВИ, СУПЕРВАЙЗИНГОВЕ УПРАВЛІННЯ, АВТОМАТИЗАЦІЯ, МОДЕЛЬНО-КЕРОВАНА ІНЖЕНЕРІЯ, РОБОТОТЕХНІЧНІ СИСТЕМИ, ІМІТАЦІЙНЕ МОДЕЛЮВАННЯ**

# ABSTRACT

**Master Thesis:** 78 pp., 28 fig., 2 tab., 50 sources.

**Thesis Subject:** Domain-Specific Language Methods for Supervising Generated Applications

**Object of research:** complex systems operating in dynamic environments and requiring supervisory control.

**Purpose of work:** to implement approaches, methods and models of using domain-specific languages for supervisory control of robotic systems.

**Subject of research:** methods, models and technologies of using domain-specific languages in supervisory control processes.

**Research results**

The work integrates the theory of supervisory control into the processes of developing robotic systems using MDE and improves the methods of simulation modeling of supervisory processes in complex systems.

**Conclusion**

The results of the work indicate that the use of domain-specific languages in combination with model-driven engineering and supervisory control theory is a promising direction in the development of robotic systems

**DOMAIN-SPECIFIC LANGUAGES, SUPERVISORY CONTROL, AUTOMATION, MODEL-DRIVEN ENGINEERING, ROBOTIC SYSTEMS, SIMULATION MODELING**

**ЗМІСТ**

## ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

DSLT – Domain-Specific Language Toolkit

ROS - Robot Operating System

CIF - Compositional Interchange Format

SCT - Supervisory Control Theory

GLPC – General Language Parsing Component

ASLG – Abstract Syntax Language Generator

MDLP – Model-Driven Language Processing

GAPS – Generated Application Performance Suite

HLCS – High-Level Compiler System

IMLP – Interactive Model Language Parser

MLTG – Multi-Layer Transformation Generator

# ВСТУП

**Актуальність теми.**

Розвиток робототехніки, особливо в умовах її інтеграції у промисловість, медицину, транспорт і повсякденне життя, потребує нових підходів до управління складними системами. Сучасні робототехнічні системи характеризуються динамічністю середовища, багатокомпонентністю, необхідністю адаптації до змінних умов роботи та високими вимогами до надійності. Традиційні методи проектування та управління системами не завжди здатні забезпечити ефективність і масштабованість у таких умовах.

Доменно-специфіковані мови (DSL) пропонують ефективний спосіб формалізації й автоматизації процесів управління, забезпечуючи чітке моделювання компонентів системи та спрощуючи розробку через підвищення рівня абстракції. Їхнє використання в робототехніці дозволяє:

- зменшити витрати часу на проектування;
- полегшити інтеграцію нових функцій і адаптацію до змін середовища;
- знизити ризики помилок завдяки формалізованому підходу.

Теорія супервайзингового контролю, яка лежить в основі запропонованих підходів, забезпечує управління коректністю роботи системи шляхом перевірки станів і відповідності заданим правилам. Інтеграція цієї теорії з DSL дозволяє створювати гнучкі, адаптивні та надійні рішення, що відповідають вимогам сучасної робототехніки.

У контексті світових тенденцій до автоматизації, інтелектуалізації та цифровізації систем управління, дослідження спрямоване на вирішення актуальних науково-технічних задач. Розробка інноваційних підходів із застосуванням DSL і теорії супервайзингового контролю сприяє підвищенню рівня автоматизації та надійності робототехнічних систем, розширює можливості їхнього впровадження в різних сферах діяльності.

Особливої важливості дослідження набуває в умовах зростання обсягів складних завдань, що вимагають високого ступеня автономності, таких як контроль у промисловій автоматизації, автономні транспортні засоби, роботизовані медичні платформи та розумні доми. Отже, запропоновані рішення мають як теоретичну, так і практичну значущість, забезпечуючи новий рівень розвитку робототехнічних систем.

**Мета дослідження** – імплементувати підходи, методи і моделі використання доменно-специфікованих мов для супервайзингового управління робототехнічними системами.

**Об'єкт дослідження –** складні системи, що функціонують у динамічних середовищах і потребують супервайзингового управління.

**Предмет дослідження –** методи, моделі та технології використання доменно-специфікованих мов у процесах супервайзингового управління.

**Задачі дослідження:**

1. Проаналізувати предметну область застосування доменно-специфікованих мов у робототехніці.

2. Визначити ключові аспекти модельно-керованої інженерії (MDE) та теорії супервайзингового контролю в контексті управління роботами.

3. Розробити підхід до використання доменно-специфікованих мов у процесах супервайзингу робототехнічних систем.

4. Реалізувати імітаційне моделювання роботи робототехнічних систем для перевірки запропонованих методів і моделей.

**Методи дослідження**

В роботі використано аналіз і синтез для вивчення предметної області та побудови моделей; формальні методи для розробки правил супервайзингового управління; моделювання та експериментальні дослідження для перевірки працездатності та ефективності підходів.

**Наукова новизна отриманих результатів**

Запропоновано підхід до використання доменно-специфікованих мов для супервайзингового управління робототехнічними системами та

розроблено концепцію автоматизованої генерації артефактів на основі DSL, що спрощує проектування систем управління.

**Практичне значення результатів**

Результати дослідження можуть бути використані для проектування та розробки робототехнічних систем із підвищеною адаптивністю та надійністю і для навчання фахівців у галузі робототехніки для роботи з модельно-керованими підходами та доменно-специфікованими мовами.

**Структура магістерської роботи.** Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 78 сторінок, і містить 28 рисунків, 2 таблиці, список використаних джерел із 50 найменувань.

# РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ЗАСТОСУВАННЯ ДОМЕННО-СПЕЦИФІКОВАНИХ МОВ

## 1.1. Представлення області дослідження

Існує багато різних застосувань для робототехніки. Роботів можна зустріти в таких місцях, як заводи, лікарні та будинки престарілих. Дуже важливо, щоб ці роботи функціонували правильно та поводилися належним чином. Якщо цього не зробити, це може спричинити серйозні проблеми з безпекою, оскільки ці роботи часто працюють у динамічному середовищі з людьми. Цикл розробки такого робота часто починається з розробки моделі, яка перетворюється на програмне забезпечення. Потім програмне забезпечення перевіряється після -впровадження. Якщо щось виявляється невірним, модель змінюється, і процес починається спочатку.

Щоб контролювати поведінку робота, інженери можуть використовувати наглядові контролери. Ці наглядові контролери оновлюють свій стан на основі окремих подій, що надходять від різних компонентів - робота, наприклад датчиків відстані. Залежно від стану, в якому перебуває робот, наглядові контролери обмежують дозволену поведінку такого робота.
Програмне забезпечення для роботів можна розробляти за допомогою роботизованого проміжного програмного забезпечення, такого як ROS (Robot Operating System), роботизована операційна система. Це проміжне програмне забезпечення забезпечує зв’язок між різними частинами робота. Це має кілька переваг, оскільки дозволяє індивідуально розробляти роботизовані компоненти. Такий компонент (або вузол) може містити, наприклад логіку для зв’язку з двигуном або датчиком відстані.

Одним із способів розробки диспетчерського контролера є застосування синтезу контролера, як описано в теорії диспетчерського управління [43]. Поведінка та комунікація всіх різних систем моделюється як рослини (які в основному є просто автоматами). Події в цих системах

поділяються на дві групи: контрольовані та неконтрольовані. Контрольовані події – це події, які можуть бути викликані супервізором, тоді як неконтрольовані події виникають зовні. Кожна з цих установок визначає неконтрольовані події, а також контрольовані події. Обмеження на контрольовані події можна вказати за допомогою вимог. Ці вимоги обмежують події, які можуть відбутися на основі умов, які мають бути виконані для того, щоб відбулися контрольовані події.

Існуюча робота була виконана для використання інструментів, які підтримують синтез диспетчерського контролера (наприклад, CIF [29], більше інформації можна знайти в наступному розділі), перетворити супервізор на код і створити з нього контролер супервізора в ROS. Незважаючи на те, що сам контрольний контролер створюється автоматично, він все одно потребує індивідуального моделювання всього зв'язку, створення вузла ROS вручну та підтримки зв'язків між робототехнічним проміжним програмним забезпеченням і диспетчерським контролером в актуальному стані. Цей процес може зайняти багато часу, повторюватися та бути схильним до помилок.

У рамках цієї роботи підхід з [29] робить крок далі, використовуючи модельно-керовану інженерію. Мета полягає в тому, щоб створити доменно-орієнтовану мову (DSL), яка дозволить користувачам мови визначати зв'язок проміжного програмного забезпечення, моделювати різні компоненти та вказувати вимоги, щоб витягти з нього контролер контролю. Використовуючи DSL, ідея полягає в тому, щоб користувачі витрачали менше часу та не допускали помилок під час створення диспетчерського контролера, оскільки контролер синтезується автоматично за допомогою зовнішнього інструменту, а також генерується ROS-код із прив'язками до контролера, який запобігає помилкам.

Згенерований код підтримує кілька версій ROS. ROS — це програмне забезпечення, яке можна встановити на робота та діє як проміжне програмне

забезпечення робота. Це проміжне програмне забезпечення забезпечує зв’язок між різними частинами робота за допомогою різних засобів зв’язку.

Незважаючи на те, що існує кілька роботизованих проміжних програм [15], і різні інструменти можуть бути використані для створення диспетчерських контролерів, предметно-спеціальна мова, розроблена в даній роботі, вводить концепції з ROS (обидві версії, ROS1 і ROS2) і використовує концепції теорія наглядового контролю [43]. У цій дипломній роботі CIF, але можна використовувати будь-який інший інструмент, який підтримує синтез диспетчерського контролера.

Основна увага, а отже, і концепції DSL у цій роботі були зосереджені на проміжному програмному забезпеченні ROS, щоб забезпечити ознайомлення з концепціями ROS з точки зору зв’язку.

## 1.2. Особливості модельно-керованої інженерії` (MDE)

Model-Driven Engineering (MDE) — це методологія розробки програмного забезпечення, спрямована на підвищення сумісності між системами та спрощення процесу проектування. Модель можна визначити як абстракцію досліджуваної системи, де абстракція часто замінює модель [11]. Включаючи область проекту програмного забезпечення, керована моделлю інженерія має на меті сприяти комунікації між усіма зацікавленими сторонами в одному проекті [18]. Метою використання інженерії, керованої моделлю, є економія часу за рахунок використання концепцій ближче до домену, що підвищує продуктивність.

Перші інструменти для підтримки MDE-методології датуються ще 1980-ми роками [6]. У 90-х роках це призвело до створення Уніфікованої мови моделювання (UML), яка була прийнята як стандарт Групою управління об’єктами (OMG) у 1997 році [6]. UML дуже потужний для створення складних моделей програмного проекту. Ці моделі є єдиним джерелом істини в керованому моделлю інженерному проекті для всіх

зацікавлених сторін. Написаний код базується на цих моделях, і перевірка програмного забезпечення також відбувається з розробленої моделі.

На рисунку 1.1 зображено схему модельно-орієнтованої трансформації, яка є ключовим поняттям в інженерії, керованій моделями (Model-Driven Engineering, MDE).

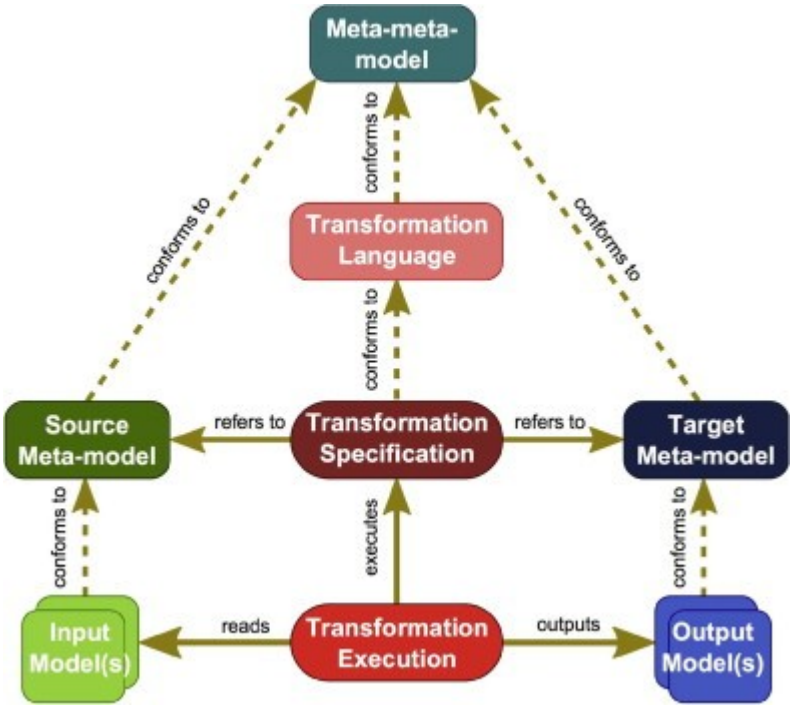


Рис. 1.1. Схема модельно-орієнтованої трансформації

Основні компоненти (рис. 1.1.):

- Мета-мета-модель: Визначає мову для опису мета-моделей. Це найвищий рівень абстракції.

- Мова трансформації: Визначає синтаксис та семантику правил трансформації.

- Вихідна мета-модель: Описує структуру та властивості вхідних моделей.

- Цільова мета-модель: Описує структуру та властивості вихідних моделей.

- Специфікація трансформації: Визначає правила перетворення вхідних моделей у вихідні, використовуючи мову трансформації.

- Виконання трансформації: Процес застосування специфікації трансформації до вхідних моделей для отримання вихідних.

- Вхідна модель: Конкретна модель, що відповідає вихідній мета-моделі.

- Вихідна модель: Модель, отримана в результаті трансформації, що відповідає цільовій мета-моделі.

Суть процесу:

1. Вхідна модель, яка відповідає вихідній мета-моделі, подається на вхід трансформації.

2. Специфікація трансформації, яка визначає правила перетворення, застосовується до вхідної моделі.

3. В результаті виконання трансформації створюється вихідна модель, яка відповідає цільовій мета-моделі.

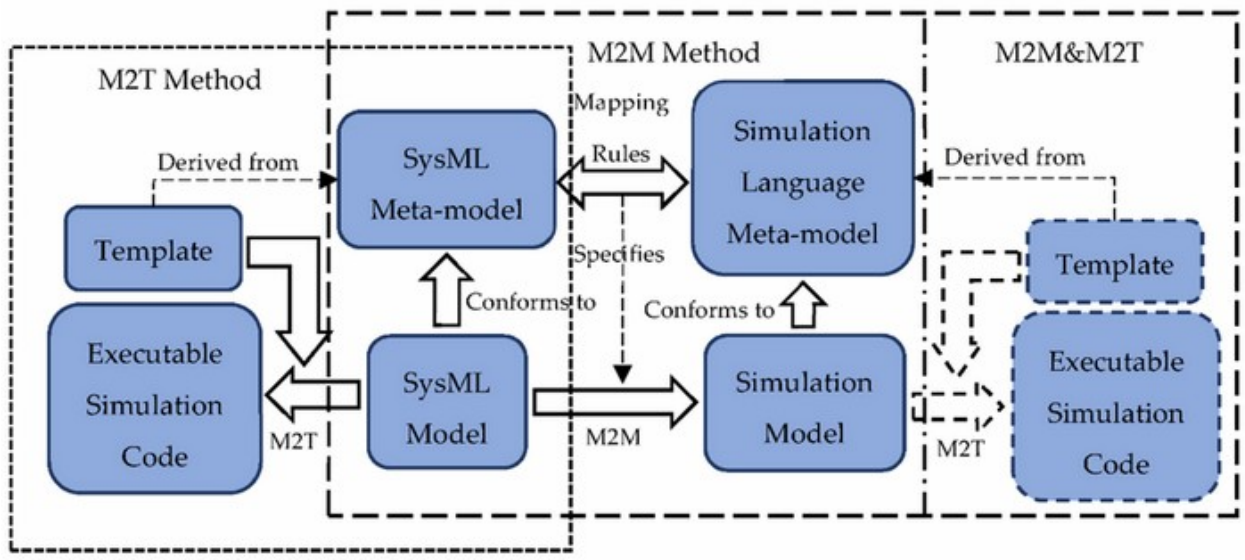


Рис. 1.2. Методи трансформації M2M і M2T

Що може зробити модельно-керовану інженерію особливо потужною, так це застосування перетворень модель-модель (M2M) або модель-текст (M2T). Ці перетворення використовуються для вилучення різних артефактів із набору розроблених моделей. Наприклад, моделі можна перетворити на код за допомогою автоматичної генерації коду як перетворення моделі в

текст. Перетворення від моделі до моделі можна використовувати для перетворення моделей у нові моделі, які можуть виступати в якості вхідних даних для іншого програмного забезпечення, наприклад формальних засобів перевірки.

## 1.3. Доменне-специфіковані мови (DSL) та операційна система робота

Мова предметної області, на відміну від мови загального призначення, як-от Java, є мовою програмування, розробленою для конкретної області. Його можна розглядати як реалізацію мови програмного забезпечення високого рівня, яка вводить концепції та абстракції, пов’язані з поточною областю. Це також обмежує свободу того, що можна робити, що може запобігти помилкам. Доменно-спеціальну мову можна використовувати для створення екземплярів сутностей у моделі домену.

Мова предметної області визначається за допомогою синтаксису та семантики. Синтаксис визначає структуру ¬мови, тоді як семантика визначає фактичне значення коду. DSL можна розділити на дві категорії: зовнішні та внутрішні мови [34]. Якщо зовнішні DSL визначають власний синтаксис і семантику, внутрішні DSL повторно використовують частину синтаксису і семантики головної мови.

Це означає, що ці мови дещо обмежені у виразності самого DSL, але вони додають можливості предметно-спеціальної мови до мови загального призначення.

Існує кілька верстаків, які можуть підтримувати розробку DSL [17]. Популярним інструментом є XText, який є фреймворком, що підтримує розробку доменних мов. Це дозволяє визначити граматику для DSL і отримати в результаті аналізатор, компонувальник, перевірку типів і компілятор. Інший варіант — JetBrains MPS який використовує інший підхід за допомогою проекційного редактора. За допомогою проекційного

редактора розробники програмують у дереві моделі, а не в конкретному синтаксисі (конкретний синтаксис — це текстове представлення синтаксису).

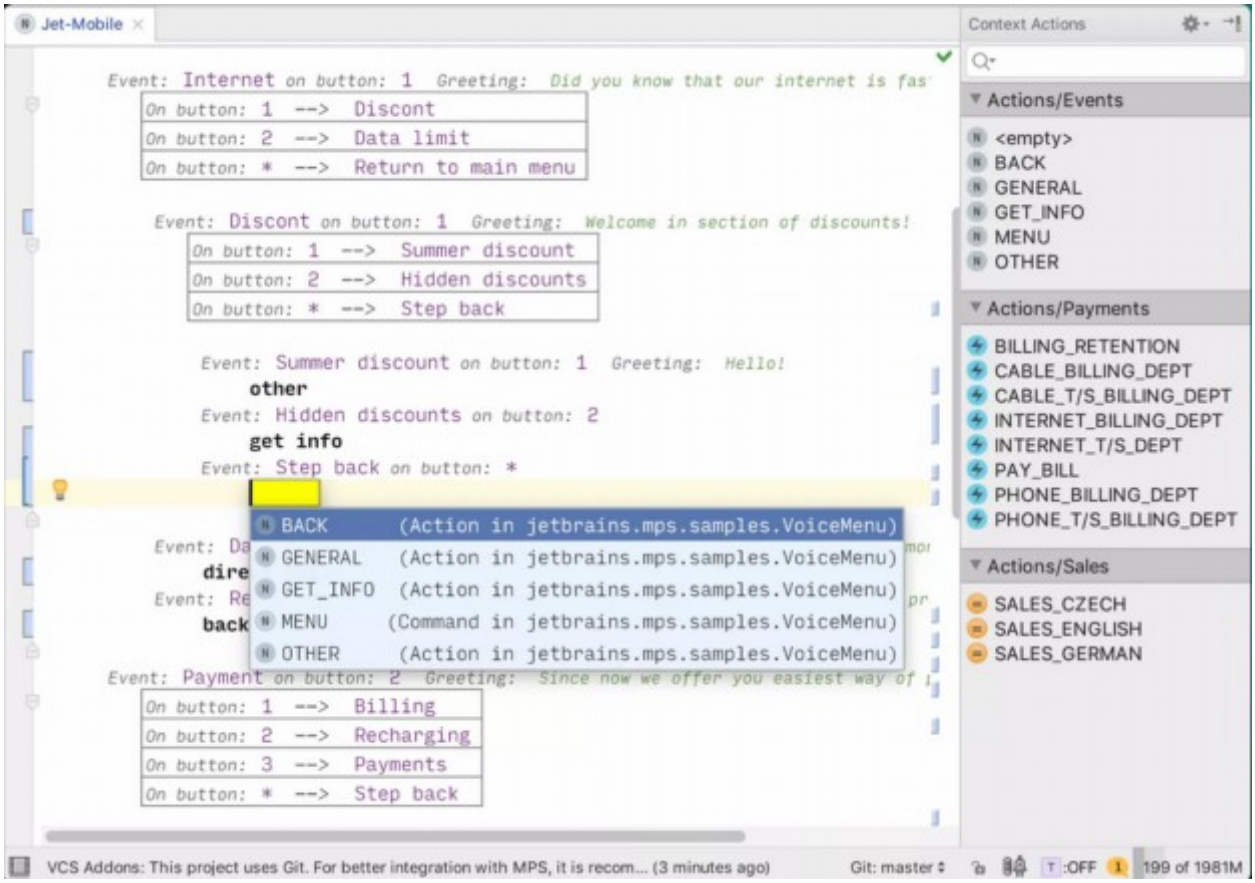


Рис. 1.3. Інструмент JetBrains MPS

Це має деякі переваги, оскільки дозволяє розробникам мови додавати складні елементи в редакторі. Наприклад, мова може представити графічне - представлення матриці або вектора.

Проміжне програмне забезпечення в робототехніці було створено, щоб полегшити створення різних процесів і уможливити зв’язок ¬між цими процесами. Роботи часто дуже асинхронні з великою кількістю апаратного забезпечення, для якого потрібні спеціальні драйвери. Загальне проміжне програмне забезпечення можна визначити як програмний клей, який забезпечує зв’язок між різними програмами в системі. Використання роботизованого проміжного програмного забезпечення може дозволити розробникам роботів зосередитися на конкретних частинах робота та

повторно використовувати існуючі рішення, оскільки частини та компоненти розробляються окремо. Існує кілька проміжних програм [15], як-от операційна система роботів (ROS), яка є єдиною проміжною програмою, яка входить до сфери цієї дисертації, щоб забезпечити ознайомлення з концепціями та комунікацією ROS.

Visual-ROS - це новий графічний веб-інтерфейс, який дозволяє розробляти програми для ROS 2 у класичному блочному інтерфейсі який побудований на базі Node-RED. Visual-ROS спрощує розробку ROS 2 для тих, хто не має досвіду програмування.

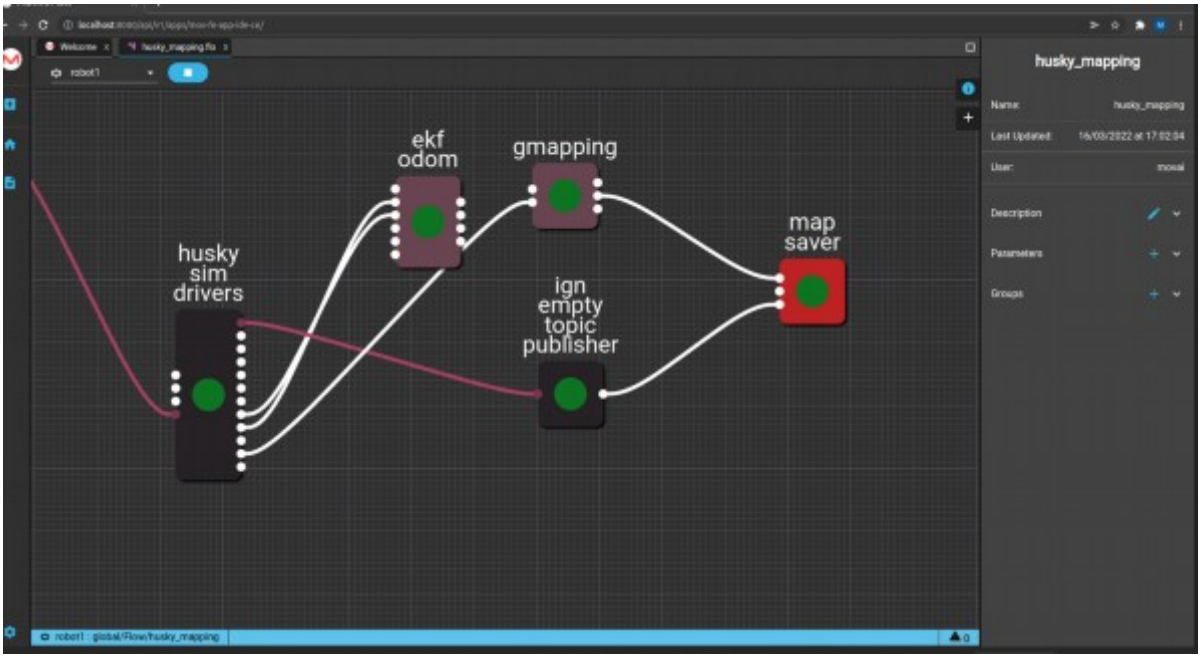


Рис. 1.4. Платформа Visual-ROS

Існують інші проміжні програми [15], наприклад Miro або Player. Miro — це розподілена об’єктно-орієнтована структура, що використовує загальну архітектуру брокера, що дозволяє розробникам створювати роботизовані програми на власній мові програмування [38]. Проект Player дозволяє створювати вільне програмне забезпечення для використання в дослідженнях і освіті [42].

Операційну систему робота (ROS) можна визначити як набір бібліотек програмного забезпечення та інструментів, які допомагають у розробці

програмного забезпечення робота. Це бібліотека з відкритим вихідним кодом, яка постачається з екосистемою, яка дозволяє повторно використовувати вже існуючі модулі, як-от найсучасніші алгоритми для навігації або драйвери для певних двигунів і датчиків. Ці модулі поширюються за допомогою Github і їх можна знайти на веб-сайті ROS. Хоча назва ROS містить операційну систему, вона працює як програма на хост-системі. Поточна версія ROS2. Попередня версія, ROS1, визначає ті самі концепції.

Основа ROS складається з графа ROS, який визначає всі вузли та зв’язок між ними. Вузол — це процес, який виконує обчислення і може спілкуватися з іншими вузлами за допомогою повідомлень, служб і дій. ROS надає клієнтські та серверні бібліотеки, які можуть полегшити це спілкування.

• Повідомлення: повідомлення мають певну структуру даних і можуть бути опубліковані в темі за допомогою моделі видавець/передплатник. Інші вузли можуть прослуховувати ці повідомлення, підписавшись на ті самі теми.

• Служби: служби базуються на моделі виклику та відповіді, подібній до моделі HTTP. Служби, які надаються вузлом, повертають відповідь, коли їх викликає клієнт.

• Дії: метою дій є підтримка довгострокових завдань. Вони складаються з мети, зворотного зв'язку та результату. Вузол може запропонувати сервер дії, до якого може підключитися вузол клієнта дії. Дію ініціює клієнт, надаючи серверу мету. Сервер визнає це та надає клієнту зворотний зв’язок під час виконання дії. Після завершення дії сервер повертає відповідь клієнту. Ціль і відповідь схожі на послугу, тоді як зворотній зв’язок надається за допомогою повідомлення.

Приклад зв'язку в ROS можна знайти на рисунку 1.5. Він складається з трьох вузлів, які спілкуються один з одним за допомогою повідомлень і

сервісів. Стрілки представляють зв’язок між двома об’єктами. Цей приклад базується на графіку з веб-сайту ROS [42].

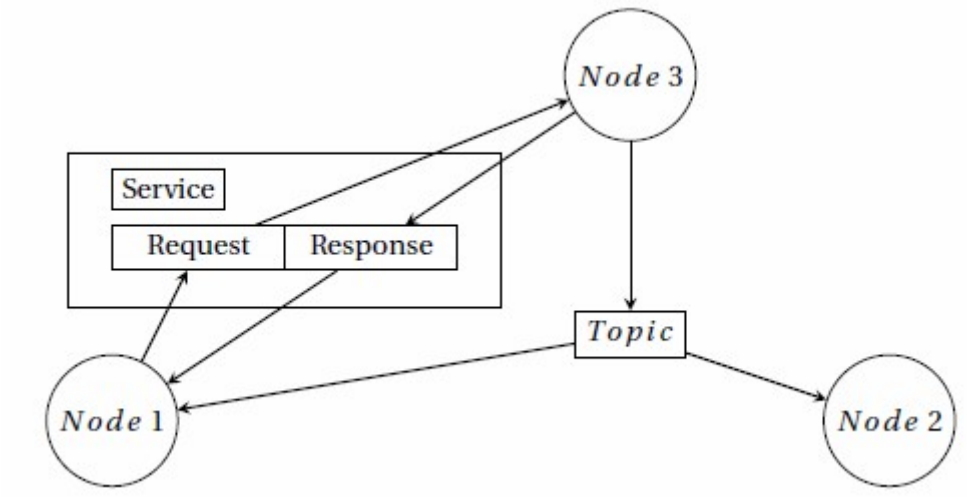


Рис. 1.5. Приклад графа ROS з трьома вузлами

Виявлення та виявлення вузлів в одній мережі відбувається автоматично в ROS2, якщо вони працюють в одній мережі. Кожне налаштування ROS2 має ідентифікатор домену, який є числом, яке використовують усі вузли для зв’язку один з одним. Поки вузли спілкуються з тим самим ідентифікатором домену, вони будуть виявлені. У ROS1 виявлення відбувається за допомогою головного вузла. Виявлення продовжується навіть після початкової фази налаштування, щоб забезпечити періодичну перевірку наявності нових вузлів. Крім того, коли вузли виходять з мережі, вони мають можливість повідомити про своє завершення роботи.

Щоб запустити програму в ROS, можна безпосередньо запустити вузол або використати файл запуску [45]. Ці файли запуску містять інформацію про те, як запустити один або кілька вузлів, що дозволяє користувачам швидко запускати набір вузлів, які керують усіма частинами робота. Він також підтримує повторне використання наявних файлів запуску, які можна використовувати для одночасного запуску всіх вузлів, необхідних для певного робота.

В даний час існує дві основні версії ROS: ROS1 і ROS2. Оновлення ПЗ для обох версій випускаються у вигляді дистрибутивів. Існують деякі відмінності між ROS1 і ROS2, як описано в [44]. Найпомітніші зміни:

• ROS2 підтримує декілька платформ (Ubuntu, MacOS і Windows) із коробки

• У ROS2 додано підтримку вузлів реального часу.

• Дії є частиною основного проекту ROS у ROS2, замість того, щоб вимагати додаткової бібліотеки.

• На відміну від версії 1 ROS, ROS2 не потребує центрального майстра ROS. Усі вузли мають можливість виявляти інші вузли без використання централізованої системи.

• Передача даних відбувається за допомогою стандарту DDS, а не спеціального протоколу [44].

## 1.4. Теорія супервайзингового контролю

Теорія супервайзингового наглядового контролю, скорочена як SCT (Supervisory Control Theory) була запропонована в [33]. Ідея полягає в тому, щоб обмежити поведінку дискретної системи шляхом синтезу супервізора. Дискретна система — це система, яка має зліченну кількість станів, на відміну від безперервної системи, яка має нескінченну кількість станів.

У теорії диспетчерського керування спочатку дають визначення поняття генератора, який схожий на те, що таке автомат. Генератор буде виконувати роль заводу, який є репрезентацією окремої системи та її станів і є об'єктом управління. Формальне визначення генератора можна знайти в [33].

Стани генератора можна позначити як помічені тобто такі що мають якесь особливе значення. Прикладом такого значення є те, що вони представляють завершене завдання.

Події можуть виникати як з установки, так і ззовні. Якщо такий процес повертається до початкового стану через деякий час, початковий стан можна визначити як маркерний стан. Щоб застосувати синтез контролера, має бути можливість досягти маркованого стану з усіх досяжних станів у системі [36].

Генератор буде керувати системою та діяти як установка. Для цього теорія наглядового контролю визначає керовані події, які також не обмежують систему більше, ніж це необхідно. Σ як множина всіх подій, і Σc як множина керованих подій, або подій, над якими контролер має контроль. Така керована подія може бути або дозволена, або заборонена. Якщо вона заборонена, контролер ніколи не дозволить події відбутися, тоді як якщо вона дозволена, вона не змушена відбуватися.

Окрім керованих подій, є також події, над якими контролер не має контролю: некеровані події. Ці події оновлюють активний стан установки та можуть відбуватися в будь-який час. Контролер спостерігає за послідовністю некерованих подій і може обмежити установку виконувати керовану поведінку.

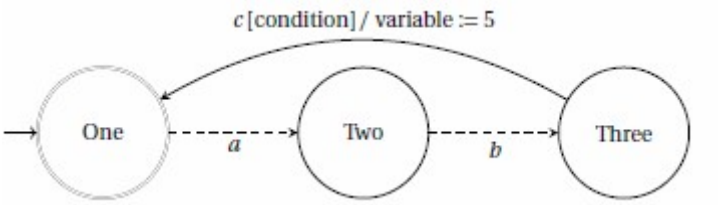


Рис. 1.6. Приклад візуального представлення установки

Установка може бути візуально представлена на діаграмі. Це можна зробити різними способами. У цій дисертації всі стани такої установки представлені колами. Якщо стан також є початковим станом, він позначається вхідною стрілкою. Подвійні кола представляють стан, який позначений. Переходи між станами візуалізуються за допомогою стрілок. Суцільні стрілки означають керовані події, тоді як пунктирні стрілки представляють некеровані події. У деяких випадках перехід (керованої події)

може мати сторожову умову, яка дозволяє перехід лише тоді, коли сторожова умова виконується. Це представлено міткою ребра, яка містить дужки навколо виразу сторожової умови. Також можливо, що перехід призводить до призначення змінних. Це представлено як суфікс до мітки ребра у формі / <змінна> := <значення>. Приклад довільної установки показано на рисунку 1.6.

Нарешті, існує поняття вимог. Вимога може вимагати, щоб установка перебувала в одному або кількох конкретних станах, щоб дозволити керовану подію. Ці вимоги використовуються для синхронізації контролера нагляду, який відповідає цим вимогам і ніколи не дозволить відбуватися керованим подіям, які заборонені однією або кількома вимогами. Таким чином, вимоги визначають поведінку, дозволену для контролера нагляду. Вимоги можуть бути використані для синтезу контролера нагляду, який: запобігає блокуванню, лише вимикає контрольовані події.

CIF розшифровується як Compositional Interchange Format [25]. Мета CIF полягає в тому, щоб дозволити моделювати системи з дискретними подіями, синхронізовані системи та гібридні системи за допомогою автоматів. Він містить великий набір функціональних можливостей, які можна використовувати при розробці контролерів, у тому числі диспетчерських контролерів. CIF підтримує наступні функції для розробки контролерів:

- Специфікація
- Перевірка
- Синтез диспетчерського контролера
- Перевірка на основі моделювання
- Візуалізація на основі моделювання
- Тестування в реальному часі
- Генерація коду

У цій роботі CIF буде використовуватися лише для виконання синтезу диспетчерського контролера та генерації коду з цього диспетчерського контролера.

## Висновки до розділу

В даному розділі проведено комплексний аналіз предметної області застосування доменно-специфікованих мов (DSL) з урахуванням сучасних підходів до розробки програмних систем. Було визначено ключові аспекти, що характеризують предметну область, де використання DSL може забезпечити ефективність у процесах розробки та експлуатації програмного забезпечення. Особливий акцент зроблено на важливості вузької спеціалізації цих мов для досягнення оптимізації в конкретних доменах. Описано концептуальні основи MDE, яка відіграє центральну роль у прискоренні процесів розробки через моделі як основний артефакт. Зазначено, що MDE дозволяє зменшити розрив між моделлю системи та її реалізацією завдяки автоматизації трансформацій і генерації коду.

Підкреслено переваги DSL у робототехніці, зокрема у створенні операційних систем роботів. Використання DSL спрощує опис поведінки, управління процесами та взаємодії компонентів, що сприяє підвищенню надійності та адаптивності систем.

Загалом, у розділі окреслено теоретичні та практичні засади застосування доменно-специфікованих мов у контексті модельно-керованої інженерії, що створює підґрунтя для подальшого дослідження.

# РОЗДІЛ 2. ПІДХОДИ, МОДЕЛІ ТА МЕТОДИ ВИКОРИСТАННЯ ДОМЕННО-СПЕЦИФІКОВАНИХ МОВ ПРОЦЕСУ СУПЕРВАЙЗИНГУ В РОБОТОТЕХНІЦІ

## 2.1. Предметно-орієнтовані мови для робототехніки

Цей розділ містить огляд робіт, пов'язаних з усіма аспектами дисертації. Він включає літературу про існуючі предметно-орієнтовані мови для робототехніки та їх підходи, методи оцінки, процес розробки та екосистему. Крім того, в розділі розглянуто роботи, присвячені синтезу наглядових контролерів в робототехніці, особливо в поєднанні з ROS.

### *2.2.1. Підходи*

Існує багато мов, які застосовують модельно-орієнтовану інженерію в робототехніці з використанням предметно-орієнтованих мов [33]. Одним з підходів, який можна застосувати, є підхід, заснований на завданнях, як це зробили В [23]. Вони розробили мову vTSL: формально верифіковану (за допомогою зовнішнього засобу перевірки моделей) предметно-орієнтовану мову для визначення завдань роботів. Це текстова мова, яка визначає поведінку робота за допомогою специфікацій різних завдань. Ці завдання потім перевіряються за допомогою окремого засобу перевірки моделей Spin.

Завдання визначаються за допомогою так званих дій. Дії визначають поведінку робота і можуть виконуватися одночасно з іншими діями на основі необов'язкового набору параметрів як вхідних даних. Крім того, дії мають можливість викликати та запускати інші дії. Цікавим моментом є те, що кожна дія визначається як набір поведінок, кожна з яких визначає, за яких умов поведінка має бути активною. Визначення самої поведінки використовує синтаксис, подібний до мови C. Ключовою частиною предметно-орієнтованої мови є піддії, які дозволяють повторно

використовувати логіку між різними визначеннями поведінки. Ці виклики піддій можуть виконуватися паралельно, якщо це необхідно.

vTSL також має вбудовану підтримку зв'язку з проміжним програмним забезпеченням, таким як ROS. Використовуючи ключові слова, такі як connect, read, write та disconnect, предметно-орієнтована мова додає підтримку підписки, прослуховування та публікації в теми в проміжному програмному забезпеченні. Основна увага в статті приділяється підтримці проміжного програмного забезпечення ROS. Підтримка синхронного зв'язку запит-відповідь реалізована за допомогою ключового слова query. В ROS зв'язок відповідає типу даних. vTSL дозволяє визначати власні структури даних в межах мови.

Мова реалізована за допомогою Jetbrains MPS 2, інструменту розробки мов, який підтримує створення предметно-орієнтованих мов. Однією з унікальних особливостей MPS є його проекційний редактор. Проекційний редактор підтримує редагування файлів з використанням нетекстового синтаксису. Наприклад, MPS дозволяє записувати математичні представлення векторів безпосередньо в коді. Це досягається шляхом збереження всього файлу як абстрактного синтаксичного дерева, а не простого текстового представлення. Проекційний редактор підтримує лише введення команд, які відповідають граматиці мови, що виключає синтаксичні помилки.

Дії та поведінка мови перетворюються в код за допомогою перетворення "модель-в-текст" (M2T). Однак у статті не описано методологію цього перетворення.

Одним із випадків використання побудованої моделі поведінки робота є виконання перетворення "модель-в-модель" (M2M) в модель для засобу перевірки моделей Spin. Модель перетворюється на кінцевий автомат за допомогою заглушок компонентів. Оскільки мова не має поняття про можливі стани та взаємодії вузла проміжного програмного забезпечення, їх також необхідно моделювати. Тому вводять рівень навичок, який моделює

теми та сервіси проміжного програмного забезпечення. Ці визначення не обов'язково відповідають одному вузлу ROS, але можуть описувати поведінку групи вузлів.

Автори вирішили дозволити користувачеві визначати твердження про значення змінних в межах предметно-орієнтованої мови, а не використовувати для цього окремий інструмент. Це дає деякі переваги, оскільки змінні можна легко зіставити зі станами в результуючій моделі. Однак твердження, які можна виконати, все ще дуже обмежені. Іншим недоліком vTSL є те, що все ще досить нечітко та невизначено, як логіка програми має бути розділена на різні дії, поведінки чи функції. Крім того, зв'язок з проміжним програмним забезпеченням призводить до використання неоднозначних ключових слів. Нарешті, в реалізації, описаній у статті, все ще відсутній підхід до генерації коду.

Приклад програми, написаної мовою vTSL, наведено в лістингу 2.1. Зверніть увагу, що рівень навичок та типи даних ROS-мосту були опущені.

Лістинг 2.1. Приклад робота, який рухається до зіткнення з перешкодою, написаний мовою vTSL

```
action MoveUntilObstacle(double speed)
    behavior normal
        connect DistanceSensor.distanceTopic as distanceSensor with queue size 1;
        DistanceMessage curMsg = read distanceSensor;

        while (curMsg.distance < 100)
            curMsg = read distanceSensor;
            setSpeed(speed)
        end on abort
            disconnect distanceSensor;
            setSpeed(0);
        end

        disconnect distanceSensor;
        setSpeed(0);

        return;
    end

    function setSpeed(double speed)
        Float64 speedMsg;
        speedMsg.data = speed;

        write speedMsg to Motor.speedTopic;
    end
end
```

Інший текстовий підхід був запропонований у [46], а саме MontiArcAutomaton. MontiArcAutomaton - це фреймворк, який дотримується архітектури "компонент і з'єднувач" (C&C). У цій архітектурі композиція всіх компонентів відокремлена від індивідуальної поведінки окремого компонента. Зв'язок між цими компонентами визначається як порти. Поведінка компонента може бути визначена як автомат або за допомогою проблемно-орієнтованої мови. Підтримка цих розширень розроблена в MontiArcAutomaton ADL.

Програмна архітектура робота з дослідження [46] зображена на рис. 2.1. Компонент Controller отримує дані про відстань та колір тосту від підключених датчиків. Автомат I/Oω, що моделює поведінку Controller, перетворює ці вхідні дані на команди для ToasterController, який запускає та зупиняє сам тостер, та компонента ArmController, який керує роботизованою рукою для захоплення та доставки тосту. Поведінка компонента ArmController моделюється як набір програм RA.

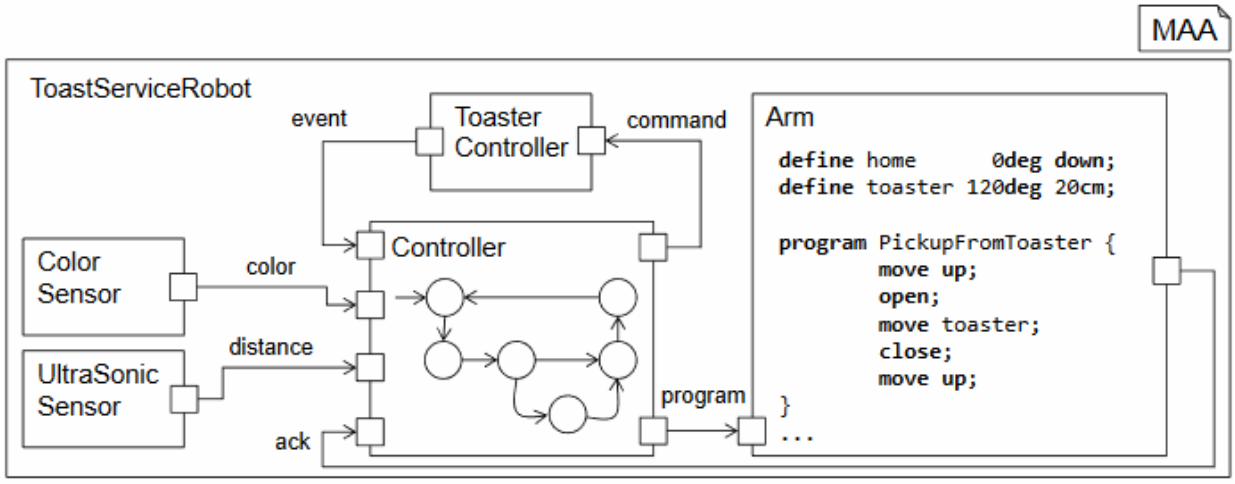


Рис. 2.1. Архітектура ToastServiceRobot із вбудованими програмами RobotArm

Щоб згенерувати виконуваний код з архітектури, інженер-програміст повинен надати генератор коду для вбудованої мови RA, який перетворює команди RA на код для цільової платформи. Цей генератор коду може бути

обраний з бібліотеки існуючих генераторів коду або розроблений заново. Нарешті, генератор має бути інтегрований у фреймворк таким чином, щоб він виконувався щоразу, коли обробляється компонент з програмами RA.

Мова MontiArcAutomaton базується на інструменті MontiCore Language workbench. Мова визначається як контекстно-вільна граматика, яку MontiCore використовує для генерації парсера, який може розібрати конкретний синтаксис в абстрактне синтаксичне дерево (AST). Хоча мови MontiCore є текстовими, передбачена інтеграція з Eclipse Modeling Framework (EMF) для підтримки візуальної розробки моделей.

MontiCore підтримує композицію мов на основі трьох принципів:

- Вбудовування: використання частин існуючих мов у заздалегідь визначених точках розширення.

- Агрегація: об'єднання кількох незалежних мов в одну мовну сім'ю.

Розширення: повторне використання та розширення існуючих граматик.

В роботі [46] показано можливості композиції шляхом вбудовування предметно-орієнтованої мови для роботів-маніпуляторів, що дозволяє моделювати поведінку компонента, який керує маніпулятором. Зв'язок відбувається шляхом визначення одного або кількох портів.

Крім того, MontiCore workbench також має функціональність для підтримки розробки генераторів коду. Ці генератори коду визначаються в конфігурації програми та можуть генерувати реалізації для різних мов загального призначення.

Мова MontiArcAutomaton дуже добре визначена та побудована на міцній основі. Композиційна природа мови робить її дуже придатною для широкого кола застосувань. Однак вихідний код мови доступний лише частково. Вихідний код MontiCoreAutomaton доступний в Інтернеті, але вихідний код розширюваного MontiCoreAutomaton ADL - ні. Крім того, не зовсім зрозуміло, з чого починати розробку робототехнічної програми з MontiCoreAutomaton, що робить поріг входження для розробки з використанням цієї предметно-орієнтованої мови відносно високим.

Лістинг 2.2 містить приклад контролера робота, який рухається до зіткнення з перешкодою

```
component Controller {
    port
        in int distance,
        out int speed;

    automaton {
        state Moving, Halt;

        Moving -> Halt {distance < 100} / { speed = 0 },
        Moving -> Moving / { speed = 100 };
    }
}
```

Графічний підхід був застосований у [13]. Цей підхід був вперше опублікований у 2012 році, але він все ще має деякі цікаві підходи, які варто розглянути. Підхід базується на Eclipse Modeling Framework (EMF). EMF - це розширення для Eclipse, яке полегшує розробку моделей та генерацію коду для різних наборів інструментів та застосувань на основі структурованого набору моделей. Він також використовує Papyrus для підтримки графічного редагування моделей.

Мова була розроблена на основі набору вимог. Ці вимоги включали те, що мова має бути простою у використанні, вона має підтримувати компонентно-орієнтовані архітектури та підтримувати кілька цільових платформ за допомогою генерації коду. Автори не проводять явної оцінки простоти використання, а лише міркують про це. Вони стверджують, що мова не вимагає від користувачів знань програмування, а деталі платформи приховані в моделях.

RobotML починається з проектування моделі, яка представляє сенсори, актуатори та системи управління різних роботів. Зв'язок визначається як порти та з'єднувачі. Як архітектура "видавець-підписник", так і механізми "запит-відповідь" підтримуються через ці порти. Поведінка систем управління моделюється у вигляді кінцевих автоматів, які представляють різні стани кожної системи та використовують вхідні порти та вихідні з'єднувачі. Наступним кроком розробки програми є визначення плану

розгортання, який визначає використовуване проміжне програмне забезпечення для роботів.

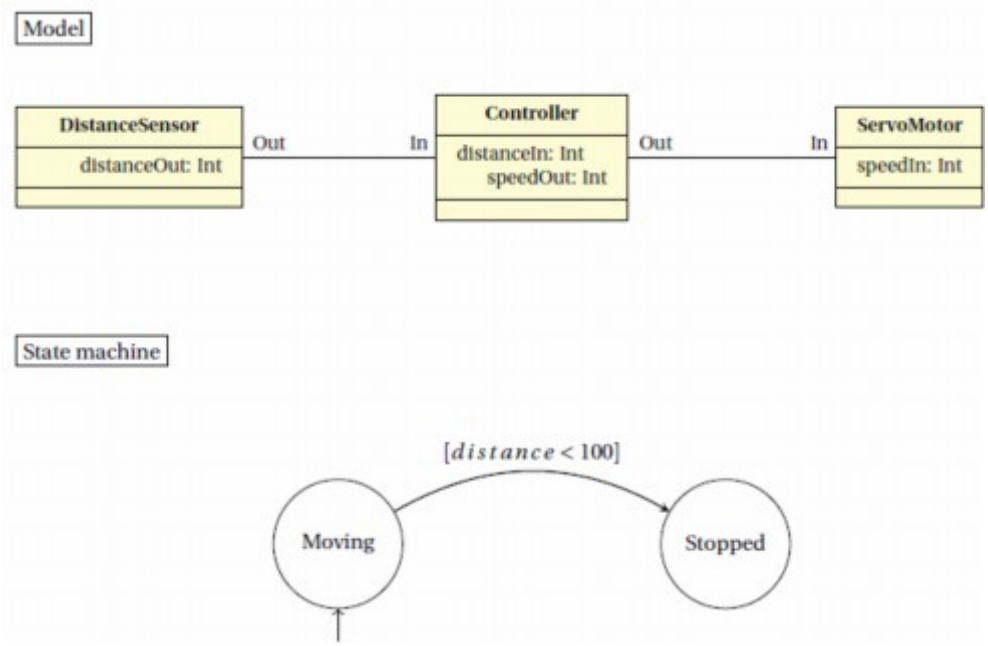


Рис. 2.2. Ескізний приклад робота, який рухається до зіткнення з перешкодою, в RobotML

Модель RobotML була заснована на онтології робота. Для розробки мови було проведено дослідження області робототехніки. Серед визначених концепцій - робот, сенсорна система, система актуаторів та система локалізації.

Дослідження про RobotML була опублікована ще в 2012 році, що може бути причиною проблем, які виникають при спробі встановити RobotML в новіших версіях Eclipse та Papyrus. Остання активність на сторінці організації RobotML на GitHub датується 2016 роком, що може свідчити про відсутність підтримки фреймворку. Крім того, мова має опції для розгортання на одному або кількох проміжних програмних забезпеченнях (для роботів), але все ще вимагає ручного визначення зв'язків з платформою.

Як зазначено вище, RobotML застарів і більше не сумісний з новими версіями Eclipse та Papyrus. Ескіз рішення в RobotML зображено на рисунку 2.2.

Інший підхід, розроблений у [16], полягає у використанні синтезу для вилучення контролера. Це використовує підхід "правильний за визначенням", на відміну від ручного підходу до побудови контролера робота, який коректно (на основі специфікацій) реагує на зміни в динамічному середовищі, оскільки контролер витягується за допомогою синтезу. Ручний процес може бути дуже трудомістким та схильним до помилок. Мова використовує специфікації Generalized Reactivity(1) (скорочено GR(1)) для автоматичної генерації проектів контролерів з специфікації. У статті пропонується Salty: предметно-орієнтована мова, яка дозволяє визначати специфікації GR(1).

Ці системи GR(1) працюють наступним чином. Специфікація φ має вигляд φ = φe =⇒ φs, де φe містить припущення про середовище, а φs кодує гарантії, які система повинна забезпечити за цих умов середовища. Мова дозволяє користувачам створювати ці специфікації та використовує зовнішні інструменти для застосування фактичного синтезу цих специфікацій GR(1). Предметно-орієнтована мова додає підтримку складних або повторюваних виразів за допомогою загальних шаблонів та макросів. Синтезований контролер (який є математичним представленням контролера) перетворюється на кілька мов загального призначення, таких як Python, Java та C++.

На верхньому рівні Salty визначає поняття контролера, який приймає вхідні та вихідні дані, визначені на контролері. Ці типи вхідних та вихідних даних можуть бути переліками, які представлені як бітові вектори. Властивості специфікації можна додавати у відповідний блок.

Ініціалізація системи та ініціалізація середовища мають свій власний блок, те саме стосується властивостей переходу в системі та середовищі, а також властивостей живучості як для системи, так і для середовища. Крім того, Salty додає підтримку таких функцій, як перевірка справності (приблизна оцінка того, чи є специфікації реалізованими), налагодження та оптимізація.

*2.1.2. Оцінка мови*

Оцінка мови є ключовим фактором при прийнятті рішення про те, чи підходить мова для даної задачі, не лише з боку користувача мови, але й з боку розробника мови. Автори показують, що оцінка проводиться або за допомогою кількісної, або якісної оцінки [33]. Якісні оцінки зосереджені на продуктивності мови з точки зору часу виконання або часу збірки, тоді як кількісні оцінки більше зосереджені на тому, наскільки добре концепції можуть бути змодельовані за допомогою підходу предметно-орієнтованої мови.

Деякі статті вибирають оцінку можливостей своєї мови, дозволяючи їй виконувати спеціалізоване завдання та виконувати його на різних фізичних платформах [25, 44, 49]. Під час тематичного дослідження були обрані різні типи роботів. У деяких випадках ці роботи мали різні сенсори та актуатори, щоб підкреслити обробку роботів з різними можливостями.

Є також наукові дослідження статті, які покладаються виключно на оцінку в симуляторі [45], що має деякі переваги з точки зору відтворення, але все ж було б краще побачити приклад з реального світу. В інших випадках оцінка взагалі не проводилася, а скоріше згадувалася як точка покращення [16].

Інший підхід до оцінки мови був застосований в [41]. Для оцінки простоти мови автори продемонстрували мову за допомогою живої демонстрації аудиторії та застосували запропоновані користувачами зміни за короткий проміжок часу.

Окрім якісних підходів до оцінки, існують також кількісні методи. Прикладами таких метрик є метрика "продуктивність на ват" [49] або час, який потрібен кільком групам для вирішення заданого завдання за допомогою мови [27]. Є також статті, які оцінюють переносимість мови. Існує два визначення переносимості. По-перше, це переносимість з точки зору можливості мови працювати на різних робототехнічних платформах.

### *2.1.3. Артефакти*

У контексті проектування, керованого моделлю, існують різні види артефактів, які генеруються з моделі. Найпоширенішим підходом, принаймні в робототехніці, є генерація коду з моделі [33]. Модель подається на генератор, який генерує один або декілька файлів із кодом. Використання кількох генераторів коду ще більше підкреслює потужність інженерії, керованої моделлю, оскільки артефакти для кількох мов або інструментів можуть бути згенеровані лише з одного джерела моделі [47].

Як зазначалося раніше, найпоширенішим сценарієм використання моделі є генерація коду. Цей код може бути кодом для таких мов, як Python або Java, або навіть спеціального фреймворку, який обробляє спеціальне завдання, наприклад планування. Існує також мова Salty [16], яка створює контролер для різних мов (Python, C++ і Java) як артефакт.

Існують і протилежні підходи, коли модель є артефактом, а не джерелом. Поєднуючи інформацію, зібрану шляхом застосування статичного аналізу коду разом із моніторингом часу виконання вузла ROS, будується модель [19]. У цьому випадку модельно-керована інженерія для робототехніки використовується як доповнення, а не як окреме рішення.

У деяких документах описуються підходи, які дозволяють передавати модель в інструмент перевірки моделі. В [12] описують, як генеруються кінцеві автомати для моделі та подаються в найсучасніший інструмент визначення пріоритетів Dona. Деякі підходи дозволяють моделювати завдання у вигляді дерева завдань, після чого вони перетворюються за допомогою перетворення від моделі до моделі в моделі, які можна перевірити за допомогою таких прикладних спеціалізованих інструментів, як Promela [23].

### *2.1.4. Екосистема*

Хоча існує багато мов, запропонованих у сфері робототехніки [33], деякі з них лише надають визначення мови в статті, залишаючи потенційним

користувачам мови мало інформації про те, як почати використовувати мову. Цікаво розглянути, як виглядає екосистема навколо існуючих мов з точки зору спільноти навколо DSL, доступності документації та інструкцій для розробників, щоб легко почати роботу.

Є документи, які надають користувачам повну граматику мови [24, 27]. Надання цієї граматики дає користувачам швидке уявлення про всі можливості мови, але все ще вимагає технічних знань і більш глибокого читання всіх структурних концепцій, які визначає мова.

Деякі статті ілюструють роботу мов, розглядаючи один або кілька прикладів сценаріїв, які ілюструють поняття, визначені в моделі мови [16, 23]. Наведення цих прикладів може сприяти сприйняттю такої мови, оскільки це дає гарне уявлення про те, як слід вирішувати конкретні проблеми, і показує виразність мови. Деякі статті вибирають демонстрацію невеликих прикладів [23], тоді як інші висвітлюють більш складні сценарії [16].

У більшості випадків лише документації мови та її концепцій недостатньо, оскільки користувачам потрібні інструкції (встановлення), щоб допомогти їм розпочати роботу. Існують документи, які направляють користувачів на зовнішні сайти, такі як GitHub, щоб надати ці інструкції [16]. Деякі зберігають всю цю інформацію на окремому веб-сайті, який є загальнодоступним.

## 2.2. Контролери для супервайзингу

Основна ідея, на якій базується ця робота, була запропонована в [29], де автори застосовують підхід до інженерії, заснований на синтезі, для створення контролера дискретних подій для платформи мобільного робота. Вони описують процес застосування синтезу до контролерів дискретних подій для складних завдань навігації з використанням CIF на основі теорії наглядового контролю. Синтезований контролер розгортається за допомогою вузла ROS1, написаного на Python.

У статті вони використовують сучасні модулі навігації, які надаються пакетом ROS. Ці інтерфейси моделюються в CIF як установка і містять модулі навігації, лазерний сканер відстані та інтерфейс. Виклик модулів навігації, представлених пакетом ROS, відбувається у вигляді дій. Кожна з цих дій була змодельована як окрема установка, що містить два стани: неактивний та активний. Керовані події мети переводять установку в активний стан, тоді як некерована подія завершення та керована подія скасування повертають її в неактивний стан. Зауважте, що зворотний зв'язок про дію в цьому випадку не враховується.

З синтезованого контролера супервайзингу генерується код та розгортається відповідно до архітектури, представленої на рисунку 2.3. Супервайзор отримує некеровані події від програмних модулів через проміжне програмне забезпечення ROS, оновлює свій внутрішній стан та вибирає відповідну керуючу дію, яка представлена керованою подією.

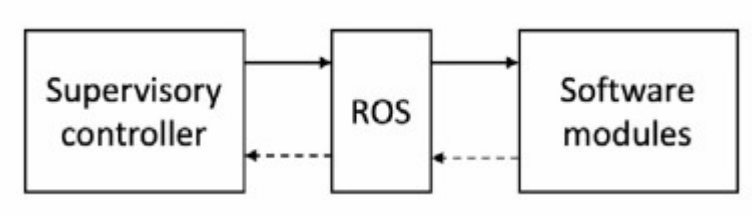


Рис. 2.3. Схематична архітектура системи

На рисунку 2.3 контролер супервайзингу вибирає відповідну керовану подію (суцільна стрілка) на основі некерованих подій (пунктирна стрілка), згенерованих програмними модулями

Окрім установок для кожної з дій навігації, автори також визначають спостерігачів, які оновлюють свій стан на основі вихідних даних модулів навігації, подій лазерного сканування та інтерфейсу. Далі в роботі визначаються вимоги, які використовуються в процесі синтезу контролера. Ці вимоги визначають умови, які потрібні керованій події, щоб бути дозволеною, або умову, яка забороняє таку керовану подію. Усі умови

визначаються як логічні вирази, які вимагають, щоб установкии перебували в певному стані. Вони визначаються з використанням як кон'юнкцій, так і диз'юнкцій.

Потім з набору заданих вимог синтезується контролер. Цей контролер синтезується за допомогою CIF, після чого викликається генератор коду CIF для генерації коду на C. Оскільки CIF не підтримує генерацію коду Python, була створена обгортка Python, яка дозволяє модулям Python 2.x взаємодіяти з контролером на основі згенерованого C-коду. Робота цього процесу генерації коду була описана в [28].

CIF3 також використовувався для наглядових контролерів поза робототехнікою, наприклад, в інженерії лінійки продуктів [3]. Вони надають приклад моделі установок кавоварки та визначають вимоги, які використовуються в синтезі, після чого синтезується наглядовий контролер.

Більше роботи було зроблено для застосування контролера супервайзингу в робототехніці [48], де вони використовують гібридний підхід і моделюють кілька роботів за допомогою моделі дискретних подій, а управління роботом - як модель на основі безперервного часу. Синтез застосовується до моделі дискретних подій, і з неї витягується наглядач.

Автори підкреслюють надійну розробку безпечних наглядачів [22], де вони поєднують найкраще з обох світів. Вони використовують як верифікований синтез, так і повне тестування. Вони пропонують робочий процес, який зводить воєдино процес синтезу, виведення тестових наборів з еталонного наглядача, генерацію коду, який виконується, систему контролю тестування та розгортання, а також інтеграцію в ширшу систему.

У [30] розглядають стан формального синтезу в галузі робототехніки. Як вони зазначають, застосування синтезу дозволяє міркувати про специфікацію завдання, а не про його реалізацію. Ручна реалізація вимагає кваліфікованих програмістів та широкого тестування для перевірки реалізації завдання. Синтез надає користувачам гарантії щодо поведінки робота.

Автори стверджують, що синтез має високий потенціал, але ще не використовується широко.

Автори представляють роботу з тестування реалізації синтезованого наглядача в [21]. Хоча можна перевірити властивості самого наглядача неможливо перевірити фактично згенерований код наглядача. Вони представляють роботу з генерації абстрактного тестового еталону замість більш складної семантики згенерованого коду. Цей тестовий еталон генерується у вигляді символічного кінцевого автомата та використовується для демонстрації еквівалентності між тестовим еталоном та згенерованим конкретним контролером, який працює на платформі системи управління.

## 2.3. Підхід застосування доменно-специфікованої мови до процесів супервайзингу

У цьому розділі буде представлено мову RoboSC (RSC). Вона дозволить розробляти контролери супервайзингу, які можна інсталювати в проміжне програмне забезпечення для роботів. Основна ідея мови базується на застосуванні інженерного підходу, заснованого на синтезі, до контролера робота, зокрема в галузі автоматичної навігації. Вони розробляють установкии теорії контролю для кожної з дій та визначають вимоги для керування їх виконанням. Після застосування синтезу контролера витягується механізм коду C. Вручну це інтегрується у вузол ROS, і зв'язки між контролером та платформою розгортання створюються вручну.

Головна мета мови полягає в сприянні впровадженню синтезу контролерів в області робототехніки шляхом надання мови, яка дозволяє синтезувати наглядові контролери з використанням концепцій теорії наглядового контролю та проміжного програмного забезпечення для роботів ROS. Завдяки наявності генерації коду користувачам не потрібно турбуватися про зв'язки та глибоке розуміння C++.

В рамках предметно-орієнтованої мови користувач може визначити всі (віртуальні) компоненти та всі комунікації, які можуть відбуватися. Подібно до термінології, якої дотримуються системи супервайзингового контролю [42], розрізняють керовану та некеровану комунікацію. Керована комунікація - це дані, що надходять від наглядового контролера до проміжного програмного забезпечення для роботів, тоді як некерована комунікація - це дані, що надходять від проміжного програмного забезпечення. На основі некерованої комунікації користувачі мови моделюють стан робота за допомогою автоматів. Потім вони можуть визначити вимоги для обмеження виконання керованої комунікації. Дані, що передаються, визначаються на основі умов, заданих користувачем. Потім з предметно-орієнтованої мови генерується код для інструменту, який може застосовувати синтез наглядового контролера, в даному випадку CIF. Код цього наглядового контролера прив'язується до вузла для проміжного програмного забезпечення для роботів, де зв'язки та код генеруються автоматично, отже, не вимагають додаткових модифікацій.

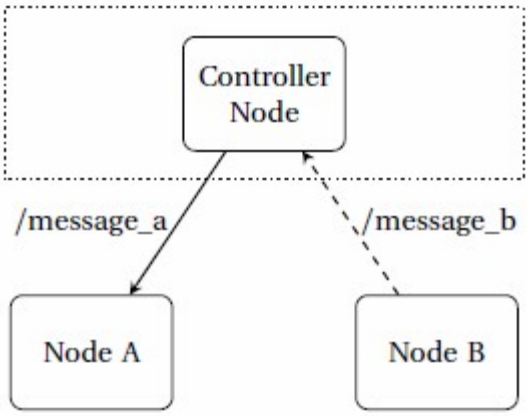


Рис. 2.4. Приклад комунікації вузла контролера супервайзингу. Пунктирний прямокутник представляє те, що генерується з предметно-орієнтованої мови.

Контролер створюється як вузол і розгортається в проміжному програмному забезпеченні для роботів. Вузол містить синтезований

наглядовий контролер та установкии, і запускає таймер, який діє як цикл управління та намагається виконати дозволену керовану подію. Набір дозволених керованих подій залежить від некерованих подій, що надходять від проміжного програмного забезпечення. Коли наглядовий контролер дозволяє здійснювати комунікацію, вузол публікуватиме повідомлення, надсилатиме запити до сервісів або цілі до дій. Щоразу, коли надходять нові дані від проміжного програмного забезпечення, стан наглядового контролера оновлюється, що оновлює внутрішній стан всіх установок. Приклад комунікації з контролером показано на рисунку 2.4.

## 2.4. Основні концепції доменно-специфікованої мови в контексті супервайзингу роботехнічними системами

Цей розділ містить всі концепції мови та те, як їх слід використовувати. Протягом цього розділу буде розглянуто наскрізний приклад, щоб зробити пояснювані концепції більш конкретними. Онтологія робота складається з двох моторів, які дозволяють роботу рухатися у двох напрямках. Крім того, він містить датчик, який може знаходити лінію, і якщо він її знайшов, він може виміряти зміщення цієї лінії відносно центру робота. Робот також оснащений датчиком відстані, який вимірює відстань до об'єкта перед роботом. Якщо є перешкода, робот може використовувати свій захоплювач, щоб підняти перешкоду. Нарешті, до робота прикріплена лампочка, яка може бути використана для відображення внутрішнього стану робота.

Мета робота - слідувати за лінією. Якщо він знаходить щось перед собою, він повинен зупинитися і підняти перешкоду за допомогою захоплювача. У випадку, якщо лінії взагалі немає, лампочка на роботі повинна активуватися.

Коли користувач мови створює модель в предметно-орієнтованій мові, він може почати або з концепції робота, або з концепції бібліотеки. Концепція робота дозволяє визначити всі компоненти, їх поведінку, їх входи

та виходи, типи даних та вимоги, яких повинен дотримуватися контролер, тоді як бібліотека може використовуватися лише для визначення всіх аспектів одного або кількох компонентів. Вони можуть бути використані повторно в рамках визначення робота для прискорення розробки різних контролерів для подібних платформ.

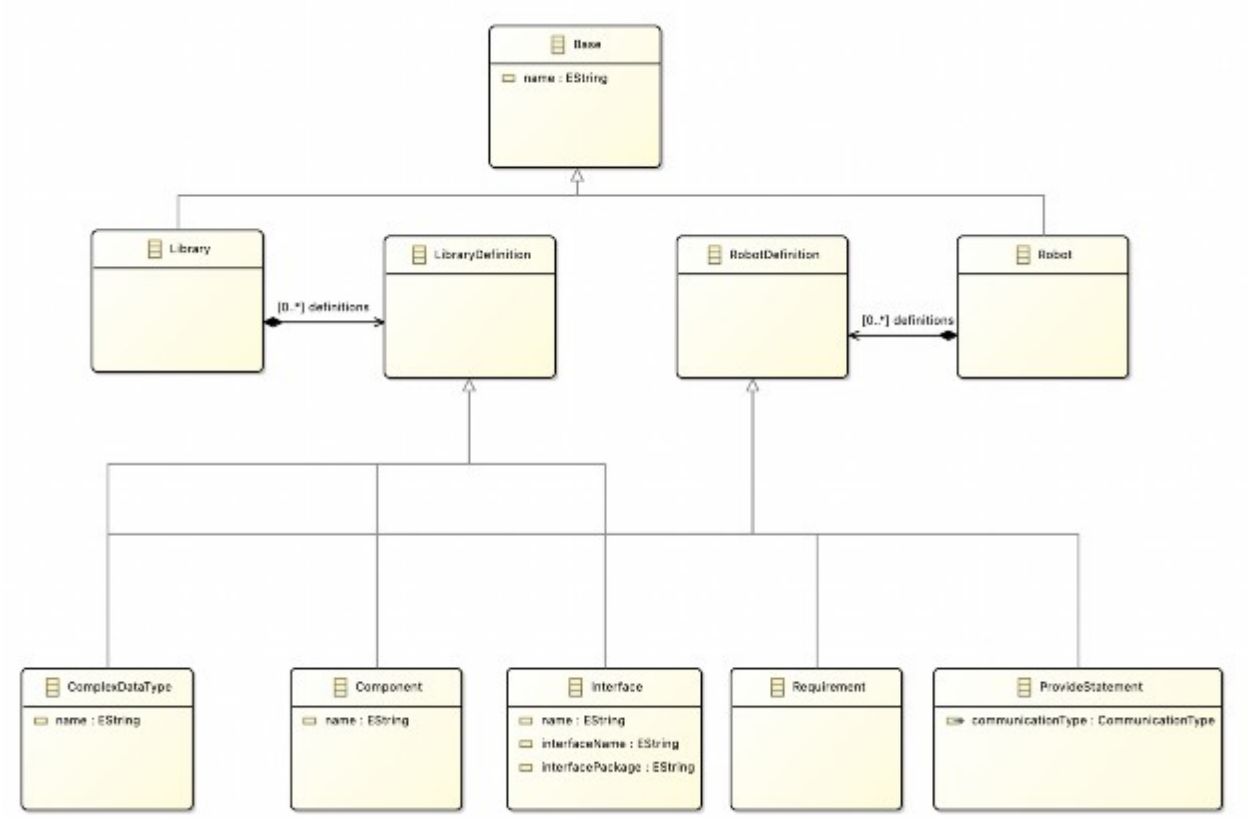


Рис. 2.5. Частина метамоделі для базових концепцій робота

Частина метамоделі бази була додана на рисунку 2.5. Вона містить концепції визначення робота та бібліотеки, які дозволяють користувачеві визначати сутності, пов'язані з бібліотекою або роботом. Бібліотеки можуть визначати лише інтерфейси, компоненти та типи даних. Сутності робота також можуть визначати оператори provide та вимоги.

Для прикладу, згаданого на початку цього розділу, слід використовувати концепцію робота. Він отримує ім'я, яке використовується для ідентифікації робота. Код, як показано в лістингу 2.3, все ще виглядає досить порожнім, але набуде форми в наступних розділах.

Лістинг 2.3. Код концепції робота для поточного прикладу

```
robot ThesisLineFollower {
    // Further robot code
}
```

*2.4.1. Компоненти*

Компонент можна визначити як віртуальну програмну абстракцію частини робота. Хоча немає явного однозначного відображення на вузол проміжного програмного забезпечення, це часто буде так, оскільки, принаймні для деяких проміжних програмних забезпечень для роботів, очікується, що "кожен вузол повинен відповідати за єдину, модульну мету" [28].

Для кожного компонента можна визначити комунікацію з проміжним програмним забезпеченням. Використовуючи цю інформацію, можна змоделювати внутрішній стан компонента. Це називається поведінкою компонента і може бути визначено за допомогою автомата. Приклад такої поведінки можна знайти на рисунку 2.6, де внутрішній стан (простого) компонента представлений двома некерованими подіями: start та stop. Ці події є керованими подіями.

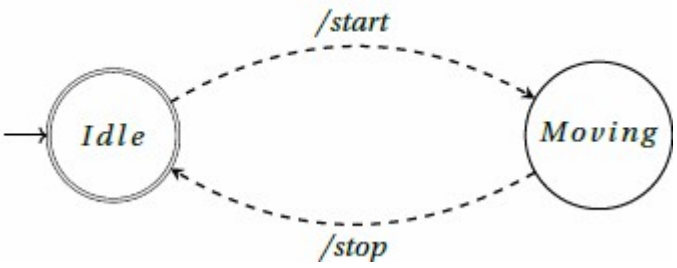


Рис. 2.6. Приклад представлення внутрішнього стану компонента двигуна

Також можна посилатися на компонент з бібліотеки за допомогою імпорту. Це називається імпортованим компонентом і не дозволяє визначати додаткові входи, виходи або поведінку. Компонент, який визначений у самому файлі, називається локальним компонентом.

Рисунок 2.7 містить концепції, що стосуються компонентів. Він підкреслює зв'язок з компонентами бібліотеки (які імпортуються) та з типами комунікації та поведінкою, які компонент може визначити через визначення компонента.

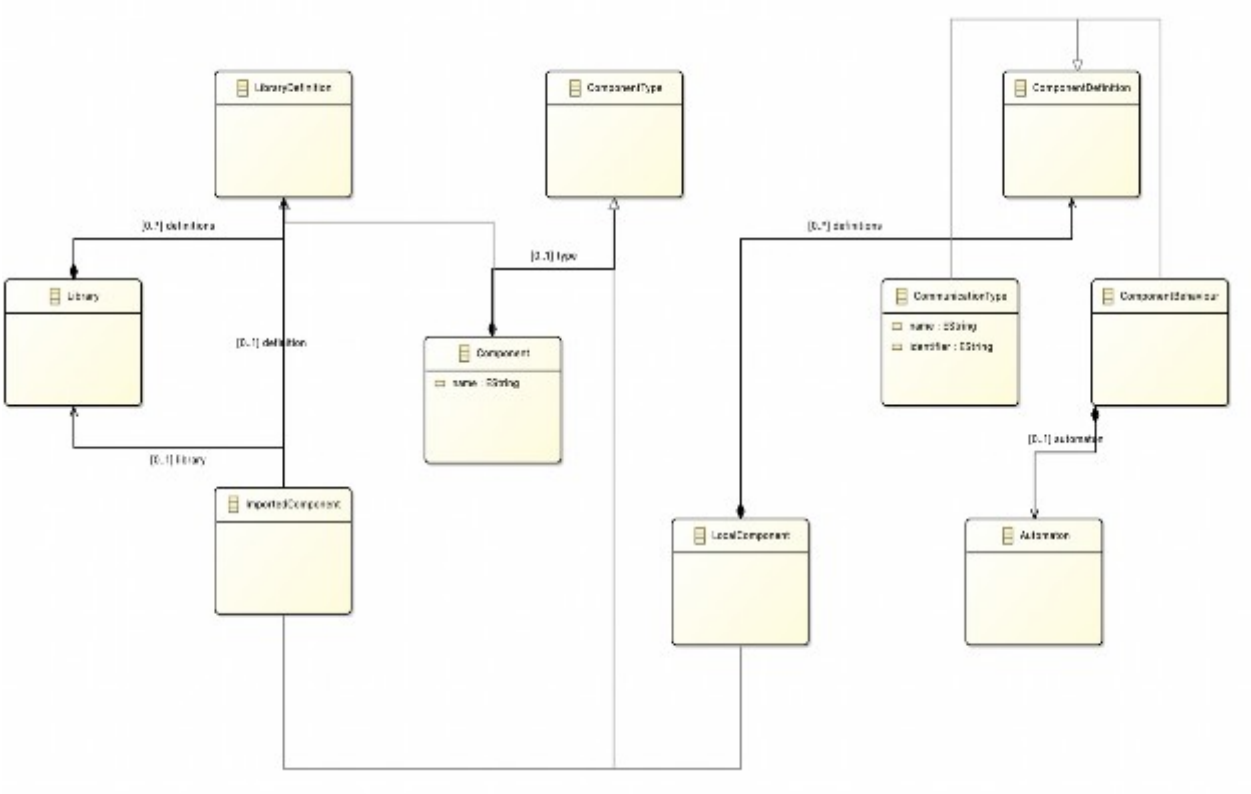

Рис. 2.7. Частина метамоделі для концепцій компонентів робота

Для робота-прикладу, який повинен слідувати за лінією, можна виділити п'ять різних компонентів. Кожен з актуаторів та датчиків моделюється як окремий компонент. Для актуаторів це означає, що є компоненти для лампочки, захоплювача перешкод та двигуна. Датчики представлені двома компонентами. Один для детектора лінії та один для датчика відстані. Цей код для цієї моделі був доданий до лістингу 2.4.

Лістинг 2.4. Частина метамоделі для концепцій типів даних робота

```
robot ThesisLineFollower {
    component DistanceSensor {}
    component ObstacleGrabber {}
    component LightBulb {}
    component LineDetector {}
    component Motor {}
}
```

*2.4.2. Процес комунікації*

Входи та виходи кожного компонента визначаються як типи комунікації. Ці типи комунікації відповідають типам підтримуваного проміжного програмного забезпечення:

- Повідомлення
- Сервіси
- Дії

Відповідні частини метамоделі додані на рисунку 2.11.

Повідомлення публікуються за допомогою анонімної моделі видавець/підписник у проміжному програмному забезпеченні. Повідомлення може бути або повідомленням, яке надходить у компонент (тобто компонент підписується на нього), або повідомленням, яке виходить з компонента (і компонент публікує його). Повідомлення має ім'я, і опціонально можна вказати тему. Якщо тема явно не вказана, ім'я повідомлення буде використано як тема. Крім того, повідомлення має відповідний тип даних, який містить формат повідомлення. Ілюстрація процесу комунікації зображена на рисунку 2.8.

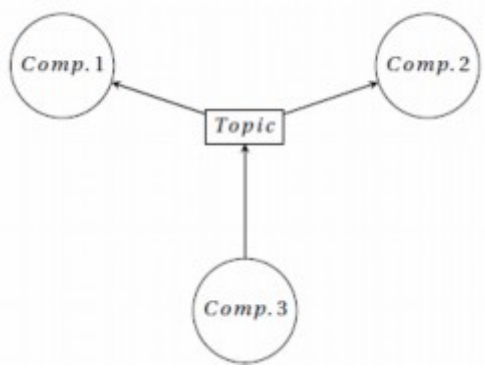


Рис. 2.8. Приклад комунікації повідомленнями між кількома компонентами

У прикладі, який був наведений на початку, є три компоненти, які визначають повідомлення. По-перше, це компонент датчика відстані, який надає наглядовому контролеру повідомлення, що містить відстань до об'єкта. Це вихідне повідомлення від компонента до контролера, і воно має тип даних enum, який вже був визначений вище. Ім'я теми визначено явно. Потім

компонент детектора лінії також визначає вихідні повідомлення. Ці повідомлення представляють значення корекції та подію, що лінії немає. Значення корекції має тип double, тоді як повідомлення про відсутність лінії не має приєднаного типу даних і тому йому призначається тип none. Нарешті, є компонент двигуна, який отримує вхідні повідомлення, що керують рухом робота.

Ці повідомлення визначаються окремо, але вони відображаються на той самий ідентифікатор проміжного програмного забезпечення. Вони відрізняються даними, які будуть надіслані разом із повідомленням. Повідомлення move рухатиме робота вперед, тоді як повідомлення stop вимагає від робота зупинити рух. Тип даних обох команд посилається на складний тип даних Twist, який був визначений вище. Крім того, повідомлення має бути пов'язане з інтерфейсом цього типу даних. Код додано до лістингу 2.5.

Лістинг 2.5. Код концепції повідомлень для прикладу запуску

```
robot ThesisLineFollower {
    component DistanceSensor {
        outgoing message distance with identifier: "/distance", type: DistanceEnum
    }

    // ...

    component LineDetector {
        outgoing message correction with identifier: "/correction", type: double
        outgoing message no_line with identifier: "/no_line", type: none
    }

    component Motor {
        incoming message move with identifier: "/vel", type: Twist links twist
        incoming message stop with identifier: "/vel", type: Twist links twist
    }
}
```

Іншим типом комунікації є сервіси. Програмна архітектура сервісу дотримується архітектури виклик-відповідь, подібно до моделі клієнт-сервер HTTP. Схематичний огляд наведено на рисунку 2.9.

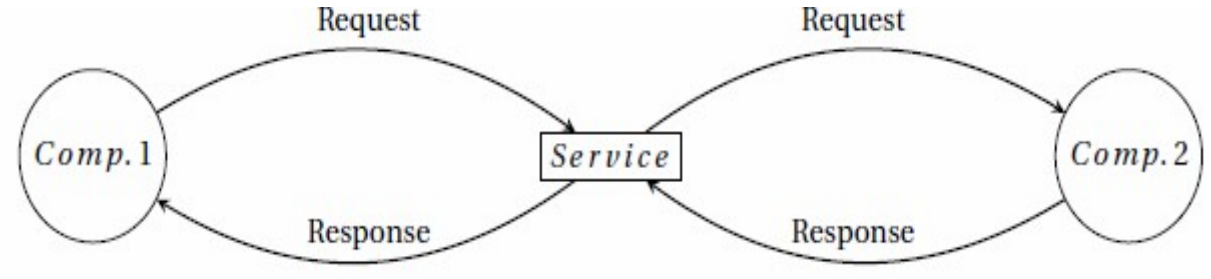


Рис. 2.9. Приклад комунікації сервісами між кількома компонентами

У прикладі є лише один компонент, який пропонує сервіс, - це лампочка. Він пропонує сервіс, якому можна надати логічне значення, що представляє стан лампочки: увімкнено або вимкнено. Якщо лампочці надіслати значення true, вона буде увімкнена, а якщо надіслати false - вимкнена. Тип даних цього запиту вже був змодельований у розділі про дані.

Відповідний інтерфейс сервісу, що містить тип даних, також пов'язаний. Код такого сервісу можна знайти в поданому нижче лістингу 2.6.

Лістинг 2.6. Код концепції сервісу для поточного прикладу.

```
robot ThesisLineFollower {
    // ...

    component LightBulb {
        service set_light_state with request: LightBulbRequest, response: none links light
    }

    // ...
}
```

Останнім типом комунікації, який можна визначити на роботі, є дії. Ці дії призначені для завдань, які виконуються протягом тривалого часу. Крім того, протягом періоду виконання дії зворотний зв'язок про процес публікується в тему, яку може використовувати ініціюючий вузол. Схематичний огляд наведено на рисунку 2.10.

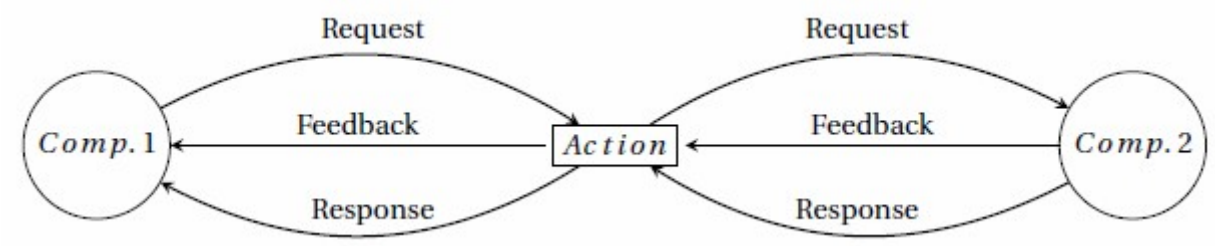


Рис. 2.10. Приклад взаємодії між декількома компонентами

Робот-приклад має дію, яку можна запустити, щоб підняти перешкоду, що знаходиться перед роботом. Він не потребує жодних даних, тому типи даних запиту, відповіді та зворотного зв'язку дії мають тип none. Потрібно, щоб дія була пов'язана з інтерфейсом, який вже був визначений, що представляє цей порожній формат дії. Визначення такої дії на компоненті можна знайти в лістингу 2.7.

Лістинг 2.7. Код концепції дії для поточного прикладу

```
robot ThesisLineFollower {
    // ...

    component ObstacleGrabber {
        action grab with identifier: "/grab", request: none, response: none, feedback: none
            links grabber
    }

    // ...
}
```

В рамках мови компоненти можуть визначати свою поведінку у вигляді автомата. Автомат має один або кілька станів і повинен мати один початковий стан. Стан також може бути позначений як маркований [43]. Має бути можливість досягти маркованого стану з усіх інших досяжних станів [36]. Кожен стан може визначати набір переходів. Це можуть бути або переходи результатів, або тау-переходи. Переходи результатів відбуваються, коли відбувається комунікація з проміжним програмним забезпеченням. Події, що запускають ці переходи, - це запити, відповіді, зворотний зв'язок та помилки. Тау-переходи можуть бути виконані, як тільки сторожова умова буде дозволена. Приклад такого автомата з вхідними повідомленнями та тау-

переходами зображено на рисунку 2.12. Зверніть увагу, що ключове слово value у призначеннях представляє дані, що надходять до проміжного програмного забезпечення.

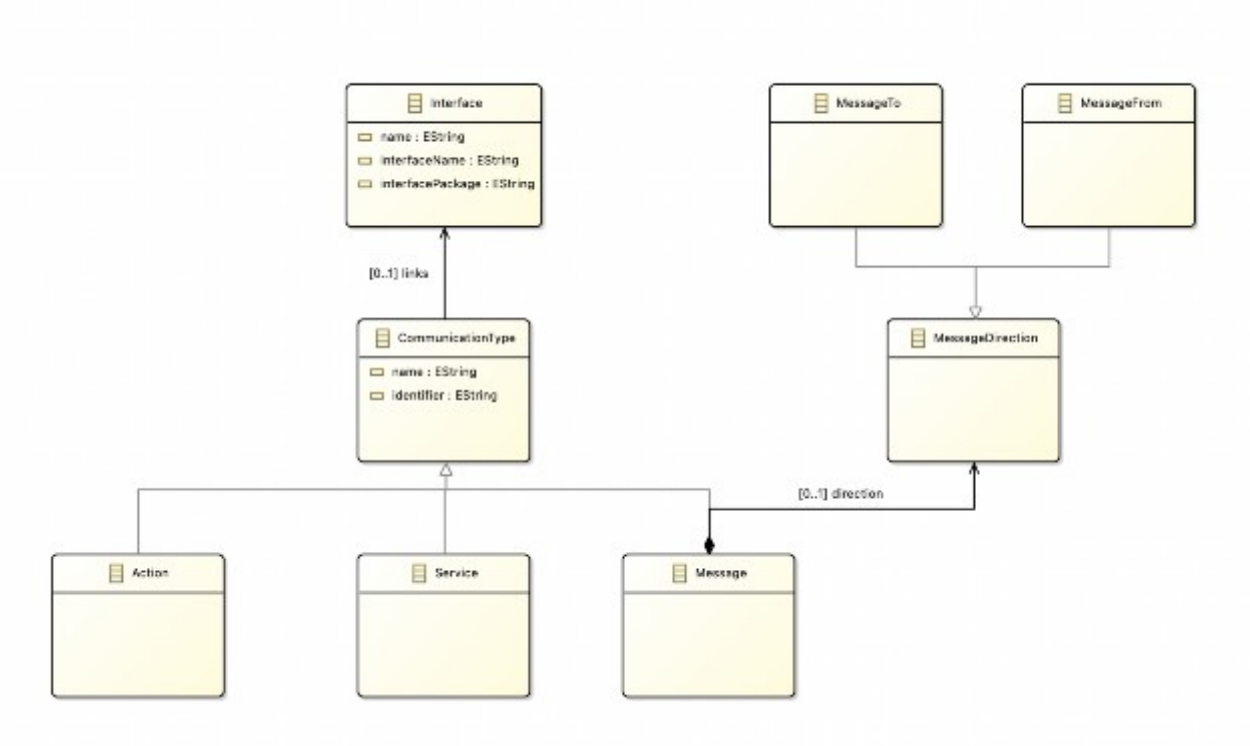


Рис. 2.11. Частина метамоделі комунікаційних концепцій робота

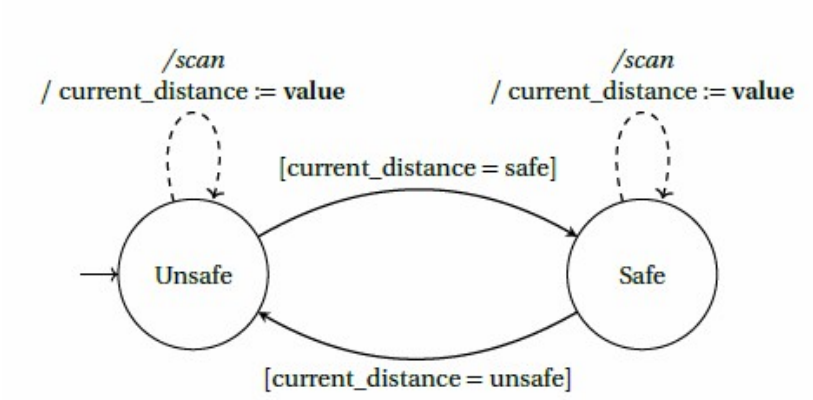


Рис. 2.12. Приклад автомата предметно-орієнтованої мови з переходами результатів та тау-переходами.

У деяких випадках користувач може захотіти мати переходи, які присутні в кожному стані автомата. У цьому випадку пропонується

можливість визначити їх на рівні автомата, а не на рівні окремого стану. Тоді перехід буде можливим у всіх станах, визначених в автоматі. Це особливо корисно у випадку, коли користувач хоче отримувати вхідні дані у всіх станах та зберігати їх.

Для зберігання даних автомати можуть визначати змінні. Змінні мають тип даних, який має бути простим типом даних. Під час переходу цим змінним можуть бути призначені значення, наприклад, на основі даних, отриманих від проміжного програмного забезпечення для роботів. Дані від проміжного програмного забезпечення представлені ключовим словом value у призначенні.

Рисунок 2.13 показує огляд концепцій, пов'язаних з автоматами в предметно-орієнтованій мові. Він містить визначення для змінних, переходів та станів.

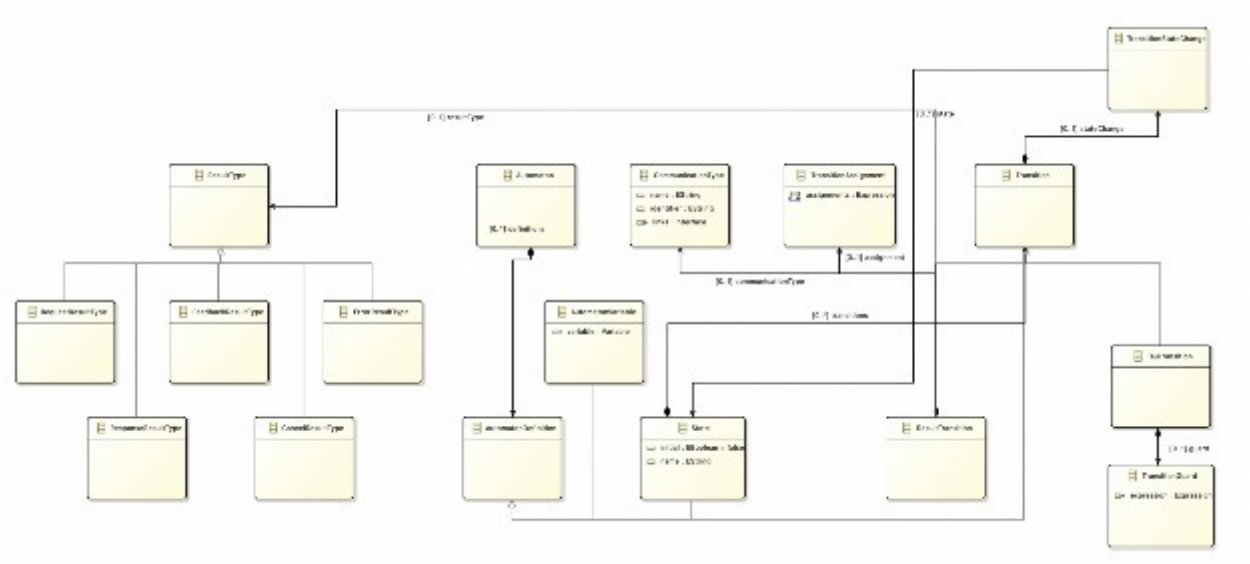

Рис. 2.13. Частина метамоделі для концепцій автоматів робота

У прикладі у вступі до цього розділу потрібно визначити поведінку трьох компонентів: датчика відстані, захоплювача перешкод та детектора лінії. Датчик відстані має лише один стан, в якому він вимірює відстані та публікує це значення. Після отримання відповіді від цього повідомлення значення зберігається у змінній. Змінна визначена в автоматі. Для

захоплювача перешкод є два стани. Перший стан, коли він неактивний, і стан, коли він захоплює об'єкт. Зверніть увагу, що неактивний стан також представляє початковий стан і є маркованим, оскільки компонент зрештою повернеться до цього стану.

Лістинг 4.8: Код концепції автоматів для поточного прикладу

```
robot ThesisLineFollower {
    // ...

    component DistanceSensor {
        // ..

        behaviour {
            variable current: DistanceEnum = obstructed

            initial marked state sensing {
                on response from distance do current := value
            }
        }
    }

    component ObstacleGrabber {
        // ..

        behaviour {
            initial marked state idle {
                on request to grab goto grabbing
            }

            state grabbing {
                on response from grab goto idle
            }
        }
    }

    component LineDetector {
        // ..

        behaviour {
            variable current_correction: double = 0.0

            initial marked state no_line {
                on response from correction goto line_found
            }

            state line_found {
                on response from no_line goto no_line
            }
        }
    }

    // ...
}
```

Коли компонент запускається, він переходить зі свого початкового стану в стан захоплення, поки не завершить захоплення об'єкта, після чого

повертається до початкового стану. Нарешті, детектор лінії також знає два стани. Є стан, коли лінії немає, і є стан, коли лінію знайдено. З початкового стану є перехід результату, який переводить автомат у стан, коли лінію знайдено. У цьому стані будь-які вхідні значення корекції зберігаються у змінній автомата. Код для цих компонентів додано до лістингу 2.8.

### *2.4.3. Надання даних*

Керована комунікація відбувається, коли виконуються певні вимоги. Це означає, що немає прямого виклику комунікації, як, наприклад, метод у звичайній мові програмування, такій як Java або C++. Тому неможливо безпосередньо передати їй дані.

Щоб вирішити цю проблему, мова вводить поняття операторів provide, які пов'язують дані з комунікацією з проміжним програмним забезпеченням для роботів. Ці оператори надають комунікацію з даними на основі логічних виразів. Ці логічні вирази визначають, чи слід надсилати дані разом із комунікацією на основі поточного стану наглядового контролера. Концепції надання даних можна знайти на рисунку 2.14.

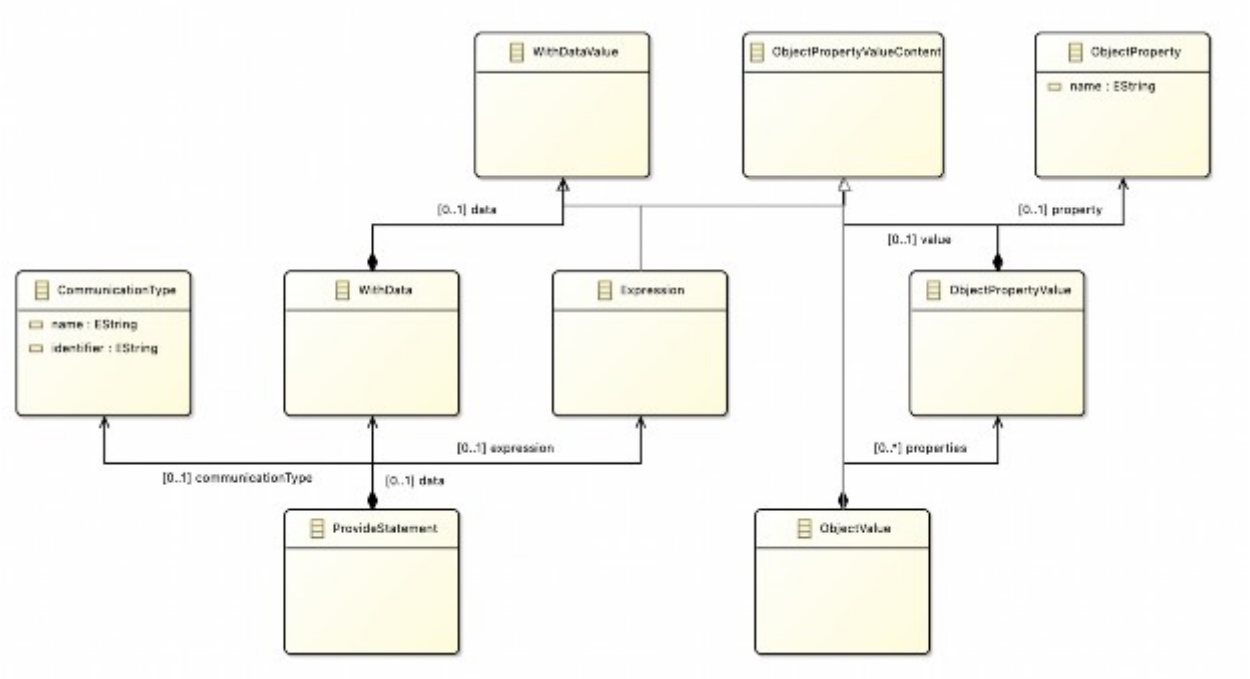


Рис. 2.14. Частина метамоделі для концепцій надання даних робота

Робот-приклад з вступу вимагає деяких операторів provide. По-перше, повідомлення, що надсилаються двигуну, вимагають лінійної та кутової швидкості. Для повідомлення move це означає, що роботу буде надано постійне значення для лінійної швидкості та частину значення корекції лінії для кутової швидкості, щоб він міг рухатися до лінії. Команда stop буде надана зі значенням нуль як для лінійної, так і для кутової швидкості. Сервіс, який оновлює стан лампочки, також вимагає значення, залежно від стану компонента детектора лінії, який додається як умова. Оператори provide наведено в лістингу 2.9.

Лістинг 2.9. Код концепції оператора provide для поточного прикладу

```
robot ThesisLineFollower {
    // ...

    // Provide communication with the correct speed
    provide move with {
        linear: { x: 0.4 },
        angular: { z: LineDetector.current / 100 }
    }
    provide stop with {
        linear: { x: 0.0 },
        angular: { z: 0.0 }
    }

    // Enable light when no line found
    provide set_light_state with { ^state: false } if LineDetector.line_found
    provide set_light_state with { ^state: true } if LineDetector.no_line

    // ...
}
```

## Висновки до розділу

В даному розділі розглянуто сучасні підходи, моделі та методи використання доменно-специфікованих мов (DSL) у процесах супервайзингу в робототехніці. Аналіз засвідчив важливість створення спеціалізованих мов для вирішення задач робототехніки. Було підкреслено, що такі мови дозволяють формалізувати специфічні аспекти роботи роботів, підвищуючи ефективність розробки та управління.

Проведено аналіз підходів до створення та оцінки доменно-специфікованих мов. Встановлено критерії оцінювання, серед яких зручність

використання, ефективність генерації артефактів, масштабованість, сумісність з іншими системами та екосистемою. Зазначено, що артефакти, створені за допомогою DSL, повинні бути інтероперабельними та придатними для інтеграції в розширену екосистему робототехнічних систем.

Запропоновано концептуальну модель використання DSL у процесах супервайзингу. Цей підхід забезпечує ефективне управління складними робототехнічними системами через формалізацію комунікацій, станів та поведінки.

Таким чином, у розділі запропоновано підхід до створення та використання доменно-специфікованих мов у процесах супервайзингу, що дозволяє підвищити функціональність та адаптивність робототехнічних систем. Це формує основу для подальшої розробки прикладних рішень у цій сфері.

# РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ МЕТОДІВ ТА МОДЕЛЕЙ ДОМЕННО-СПЕЦИФІКОВАНИХ МОВ ДЛЯ СУПЕРВАЙЗИНГУ ГЕНЕРОВАНИХ АПЛІКАЦІЙ

## 3.1. Представлення концепції генерації артефактів на основі предметно-орієнтованої мови

З предметно-орієнтованої мови генерується кілька артефактів. Ці артефакти генеруються в кілька етапів. Спочатку за допомогою перетворення "модель-в-текст" генерується код для теорії наглядового контролю, зокрема для інструменту CIF. Потім за допомогою цього інструменту застосовується синтез наглядового контролера. Після застосування синтезу можна виконати генератор коду для наглядового контролера. Цей результуючий код контролера потім використовується додатковим генератором "модель-в-текст" з предметно-орієнтованої мови, який створює вузол ROS1 та ROS2 зі зв'язками з наглядовим контролером.

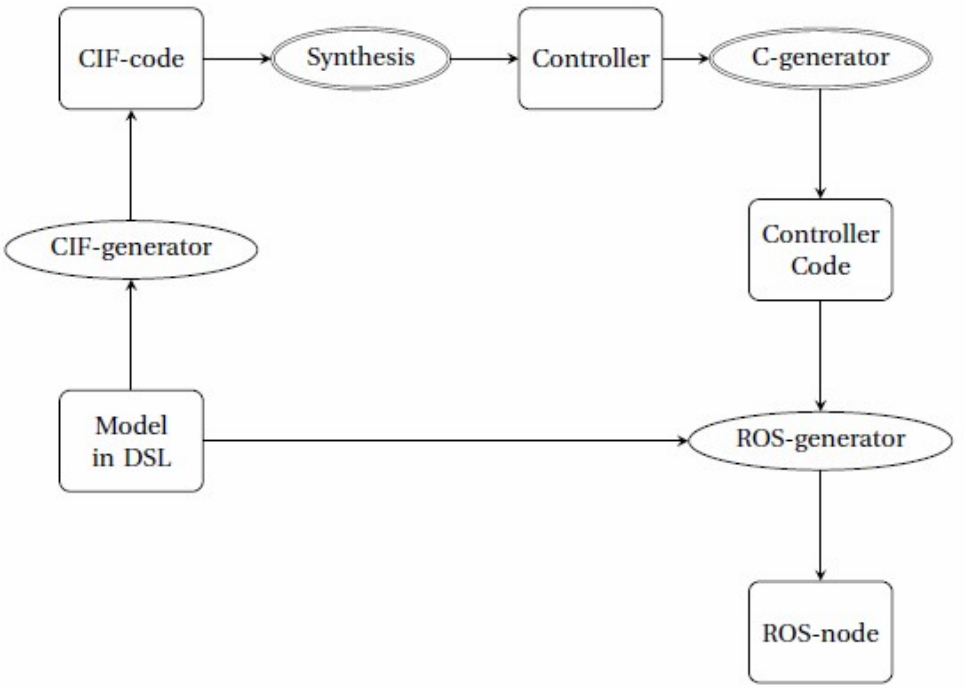


Рис. 3.1. Огляд процесу генерації артефактів на основі предметно-орієнтованої мови

На рисунку 3.1 прямокутники представляють артефакти, овали представляють генератори. Генератори з подвійними лініями позначають генератори з інструменту наглядового контролера. Стрілки передають вихід (джерело) на вхід (приймач).

На рисунку 5.1 показано процес перетворення з коду предметно-орієнтованої мови у вузол ROS. Усі прямокутники представляють артефакт, тоді як овали представляють генератори. Стрілки позначають потік даних між артефактами та генераторами, де джерело стрілки представляє вихід артефакту, а приймач визначає цей артефакт як вхід для генератора. Генератори з подвійними лініями позначають генератори з інструменту наглядового контролера.

У цьому розділі описуються згенеровані концепції та код з моделі, як для теорії наглядового контролю, так і для вузлів ROS, та те, як вони інтегруються один з одним. Крім того, визначено явне відображення між концепціями мови та концепціями ROS та теорії наглядового контролю.

### 3.2. Застосування теорії супервайзингового контролю

Серцевина контролера побудована з використанням теорії супервайзингового (наглядового) контролю (SCT). SCT використовується для застосування синтезу контролера на основі подій на основі набору установок та набору вимог. Код перетворюється на C-код за допомогою зовнішнього інструменту, який вбудовується у вузол ROS. Хоча в цій дисертації було обрано CIF, можна було б використовувати будь-який інструмент, який підтримує синтез наглядового контролера, оскільки застосовуються однакові концепції. У цьому розділі розглянуто згенеровані установкии та вимоги на основі моделі предметно-орієнтованої мови, а також показано роботу технічної інтеграції між CIF та предметно-орієнтованою мовою. Крім того, довелося внести деякі зміни до генератора коду CIF, які були запропоновані проекту з відкритим вихідним кодом.

На основі моделі предметно-орієнтованої мови будується кілька установок. У цьому розділі описуються всі згенеровані установки, їх стани, переходи та змінні.

Усі елементи комунікації представлені установками. Кожна установка визначає керовані події, некеровані події, входи, стани та ребра. Керовані події використовуються для надсилання або запуску комунікації від контролера, тоді як некеровані події представляють події, які повідомляють контролер про вхідні дані. Якщо дані потрібні в установці, входи зберігають ці вхідні дані. Залежно від типу комунікації, стани представляють фазу, в якій наразі знаходиться комунікація.

Лістинг 3.1. Алгоритм 1 для визначення властивостей входу установки для заданого типу даних

```
input : A property property (with type and name) and a prefix prefix that is prepended
        to the inputs.
output: A set of inputs that are required within CIF for a given data type.
R ← ∅
if property.type is object then
    for child in property.type do
        R ← R ∪ CIFInputs(child, prefix + _ + child.name)
    end
    return R
else
    if property required in some plant then
        return {prefix}
    else
        return ∅
    end
end
```

Установки, які підтримують вхідні дані від елемента комунікації, зберігають ці дані у входах. Для базових типів даних це призводить до однієї властивості входу. Однак для складних об'єктів для кожного поля створюється вхід. Алгоритм 1 (лістинг 3.1) знаходить всі властивості за допомогою рекурсивного пошуку та перевіряє, чи повинна властивість бути присутньою в установці, і якщо так, визначає вхід. Властивість повинна бути присутньою, якщо її значення використовується в сторожовій умові або

вимозі. Це може бути безпосередньо або опосередковано, коли значення спочатку зберігається у змінній установки. Зверніть увагу, що якщо вхід визначено в установці, саме поле може бути лише простого типу даних (переліки, цілі числа з діапазоном або логічні значення). Так, наприклад, масиви не підтримуються.

Установки, що відповідають повідомленням, відносно прості. Кожне повідомлення визначається в області видимості компонента, і залежно від напрямку повідомлення відносно компонента визначення змінюється. Для вхідних повідомлень до компонента установка має кероване ребро, яке може виникати в єдиному стані установки. Однак, якщо повідомлення виходить з компонента, в установці є одне некероване ребро. Ім'я установки - це ім'я повідомлення з префіксом message_.

Якщо дані цього повідомлення потрібні в сторожовій умові або вимозі, установка визначає входи, які зберігають вхідні дані, щоб їх можна було використовувати в сторожових умовах або призначеннях. Приклад установки для повідомлення показано в лістингу 3.2, представленому як CIF-код.

Лістинг 3.2. CIF-код для установки повідомлення

```
plant message_sample:
    uncontrollable u_response;
    input bool i_response_object_value_one;
    input bool i_response_object_value_two;

    location:
        initial; marked;
        edge u_response;
end
```

Установка сервісу виглядає трохи складніше, ніж установка повідомлення. У ROS2 сервіси асинхронні, тоді як у ROS1 вони синхронні. У нашій установці вони моделюються однаково: асинхронно. Установка розрізняє чотири різних стани: неактивний, очікування відповіді, готовий та помилка. Установка сервісу знаходиться в (початковому) неактивному стані, коли вона очікує на виконання. У цьому стані є ребро, яке дозволяє викликати сервіс, що переводить установку у наступний стан: очікування

відповіді. Коли виникає помилка, некерована подія переведе автомат у стан помилки. Якщо отримано відповідь, установка переходить у стан готовності. Подібно до того, як працюють повідомлення, результуючі значення зберігаються у вхідному полі, якщо вони потрібні в подальших сторожових умовах, вимогах призначень. Приклад установки сервісу показано в лістингу 3.3, знову ж таки представленому як CIF-код. Ім'я установки - це ім'я сервісу з префіксом service_.

Лістинг 3.3. CIF-код для установки сервісу

```
plant service_sample:
controllable c_trigger, c_reset;
uncontrollable u_response, u_error;

input bool i_response_object_value_one;
input bool i_response_object_value_two;

location idle:
    initial; marked;
    edge c_trigger goto wait_for_response;
location wait_for_response:
    edge u_response goto ready;
    edge u_error goto error;
location ready:
    edge c_reset goto idle;
location error:
    edge c_reset goto idle;
end
```

Для дій ROS установки також складаються з чотирьох станів: неактивний, виконання, готовий та помилка. Знову ж таки, неактивний представляє стан, коли дію можна запустити за допомогою ребра запуску, і якщо дія виконується, установка переходить у стан виконання. Звідси можуть виникати як некеровані, так і керовані події. По-перше, сервер дій може надавати зворотний зв'язок про виконання за допомогою некерованої події зворотного зв'язку. Це зберігає установку у тому самому стані, тоді як подія відповіді переведе автомат у стан готовності. У разі виникнення помилки (наприклад, сервер дій недоступний) виникає некерована подія помилки, яка переводить установку у відповідний стан. Ім'я установки - це ім'я дії з префіксом action_.

Як відповіді, так і зворотний зв'язок, надіслані дією, зберігаються в окремих вхідних полях установки. Приклад такої установки дії можна знайти в лістингу 3.4.

Лістинг 3.4. CIF-код для установки дії

```
plant action_sample:
    controllable c_trigger, c_reset, c_cancel;
    uncontrollable u_feedback, u_response, u_error;

    input int[0..20] i_feedback_object_value_one;
    input int[0..20] i_feedback_object_value_two;

    input bool i_response_object_value_one;
    input bool i_response_object_value_two;

    location idle:
        initial; marked;
        edge c_trigger goto executing;
    location executing:
        edge u_feedback;
        edge u_response goto ready;
        edge u_error goto error;
        edge c_cancel goto idle;
    location ready:
        edge c_reset goto idle;
    location error:
        edge c_reset goto idle;
end
```

Якщо компонент має пов'язану поведінку, ця поведінка також перетворюється на установку, де ім'я установки - це ім'я компонента з префіксом component_. Поведінка в предметно-орієнтованій мові визначається як автомат зі станами та подіями. Між станами є два можливі переходи: переходи результатів та тау-переходи.

Для переходів результатів представлене ребро в установці залежить від типу результату. Типи результатів залежать від типу комунікації та можуть мати такі значення: запит, відповідь, зворотний зв'язок, помилка або скасування.

Ребра походять від установки елемента комунікації та синхронізуються (це означає, що перехід має бути дозволений у всіх установках). Вони відповідають подіям установки, як описано в таблиці 3.1.

Таблиця 3.1.

Відображення між типами переходів результатів та відповідною подією установки

| Result Transition Type | Plant event |
|---|---|
| request | <communication plant>.c_trigger |
| response | <communication plant>.u_response |
| feedback | <communication plant>.u_feedback |
| error | <communication plant>.u_error |
| cancel | <communication plant>.c_cancel |

Якщо установка включає ребро в межах певного стану, це означає, що установка має перебувати в цьому стані, щоб ребро могло виконатися. Якщо це не так, перехід не може відбутися. Це може призвести до небажаної поведінки, оскільки проміжне програмне забезпечення для роботів завжди може надавати механізму наглядового контролера дані в будь-який момент. Якщо компонент визначає перехід, який відбувається після отримання повідомлення з певної теми, але цей перехід визначено лише в одному стані, і установка не знаходиться в цьому стані, це означає, що жодна з установок не може виконати перехід, а це означає, що система не знає про вхідні дані. Приклад такого сценарію з двома некерованими подіями a та b показано на рисунку 3.2. У лівій установці подія a дозволена в стані один, але в правій установці вона дозволена лише в стані два. Те саме стосується події b, але ця подія дозволена лише в стані два в лівій установці та в стані один у правій установці. Це означатиме, що подія не може відбутися.

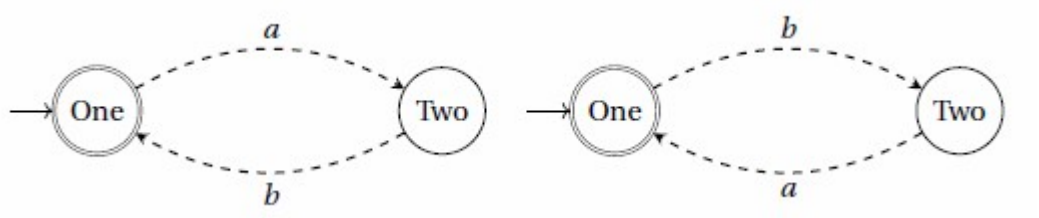


Рис. 3.2. Приклад випадку з двома установками, де некерована подія не може відбутися у звичайній теорії супервайзингового контролю

Рішення цієї проблеми можна розділити на дві частини. По-перше, важливо, щоб переходи результатів не мали сторожової умови. Якби вони мали сторожову умову, і сторожова умова не була б дозволена, це означало б, що жодна з установок не може бути повідомлена про нові дані.

Інша частина рішення полягає в тому, що кожен стан в установці поведінки компонента має бути розширений додатковими ребрами. Ці ребра не мають змін стану та сторожової умови, але гарантують, що ці події завжди можуть відбуватися, незалежно від стану установки. Для кожного стану установки (поведінки) скануються всі елементи комунікації в роботі, і всі події з таблиці 3.1 додаються, якщо вони ще не були присутні. Слід переконатися, що дублікатів немає, оскільки інакше робот може опинитися в неочікуваному стані, оскільки одне й те саме ребро визначає різні вихідні стани. Приклад показано на рисунку 3.3.

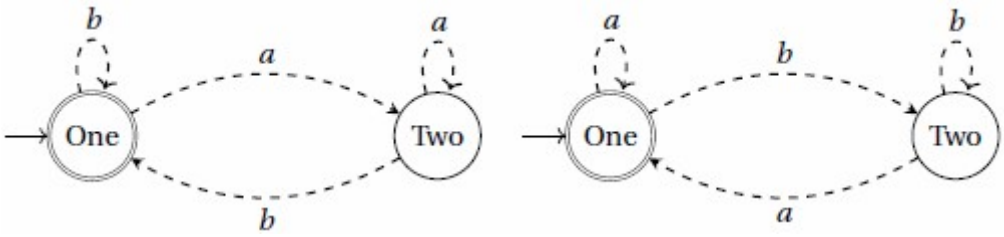


Рис. 3.3. Приклад випадку з двома установками, де некерована подія може відбутися завдяки доданим циклам

Окрім переходів результатів, автомати поведінки компонентів також можуть використовувати тау-переходи. Тау-переходи визначають сторожову умову та можливу зміну стану. Хоча CIF підтримує тау-переходи, їх не можна використовувати в синтезі наглядового контролера [36]. Щоб вирішити цю проблему, SCT-генератор генерує випадкові рядки для керованих подій. Ці керовані події визначаються на установці, і на кожній ітерації контролер намагатиметься їх виконати, якщо вони дозволені.

Деякі елементи комунікації вимагають передачі даних. Для цього було введено оператор provide. Якщо елемент комунікації має один або кілька

пов'язаних операторів provide, буде згенеровано установку, яка представляє дані, що передаються. Ця установка завжди має стан none, який вказує, що на даний момент дані не можуть бути передані. Потім для кожного оператора provide додається нове місце розташування. Ім'я цього місця розташування генерується випадковим чином. Потім кожне з цих згенерованих місць розташування містить ребра до всіх інших згенерованих місць розташування разом зі сторожовою умовою, яка була вказана у відповідному операторі provide. Є також ребро, яке повертається до місця розташування none, подія якого виконується лише тоді, коли жодна зі сторожових умов більше не виконується.

Лістинг 3.5. CIF-код для установки даних

```
plant data_sample:
  controllable c_data1, c_data2, c_none;

  location none:
    initial; marked;
    edge c_none when not (ExampleComponent.state1) and not (ExampleComponent.state2) goto none
        ;
    edge c_data1 when ExampleComponent.state1 goto data1;
    edge c_data2 when ExampleComponent.state2 goto data2;
  location data1:
    marked;
    edge c_none when not (ExampleComponent.state1) and not (ExampleComponent.state2) goto none
        ;
    edge c_data1 when ExampleComponent.state1 goto data1;
    edge c_data2 when ExampleComponent.state2 goto data2;
  location data2:
    marked;
    edge c_none when not (ExampleComponent.state1) and not (ExampleComponent.state2) goto none
        ;
    edge c_data1 when ExampleComponent.state1 goto data1;
    edge c_data2 when ExampleComponent.state2 goto data2;
end
```

Приклад установки даних наведено в лістингу 3.5, представленому в CIF-коді. Зверніть увагу, що в цьому прикладі імена ребер та місць розташування не генеруються випадковим чином. Це зроблено для зручності читання.

У цій роботі CIF використовується для застосування синтезу контролера. Концепції теорії наглядового контролю в CIF такі ж, як і концепції самої теорії, а це означає, що можна використовувати будь-який інший інструмент, який відповідає наступним вимогам:

- Інструмент підтримує теорію наглядового контролю (установки та вимоги).

- Інструмент підтримує синтез наглядового контролера.

- Інструмент підтримує генерацію коду.

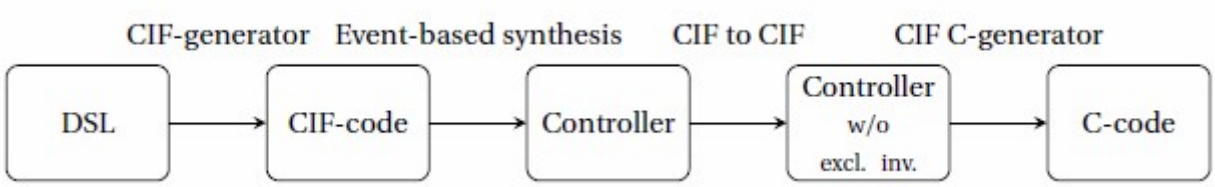


Рис. 3.4. Огляд технічної інтеграції з CIF

Після того, як генератор згенерує CIF-файл, будуть викликані класи CIF. CIF визначає кілька програм, які відповідають за операції, що виконуються над вхідним CIF-файлом. Мова використовує ці програми так, ніби вони запускаються з командного рядка, надаючи аргументи у вигляді масиву рядків. Повний процес зображено на рисунку 3.4.

У процесі використовуються три інструменти, які створюють код для контролера. Першим кроком є застосування синтезу контролера на основі подій, після чого отриманий файл перетворюється за допомогою перетворення CIF у CIF, що усуває інваріанти виключення стану/події (необхідні для генерації коду). Потім виконується генератор коду.

Однією з проблем виконання таких інструментів є те, що CIF записує вихідні дані на консоль і виходить із процесу після його завершення. Щоб запобігти цьому, вихідні дані перенаправляються та захоплюються в інший вихідний потік, а код виходу зберігається у змінній, а не зупиняється програма.

CIF підтримує генерацію коду для кількох мов програмування, включаючи C. Незважаючи на те, що згенерований код використовуватиметься в C++, код C можна скомпілювати таким чином, щоб його можна було використовувати в C++. Однак інтерфейс цього коду не

розкриває всі необхідні методи та функції, які знадобляться контролеру. Оскільки зміна згенерованого коду пов’язана з ризиком зламу коду з новішими версіями мови, зміни інтерфейсу генератора коду були реалізовані в самому CIF на основі Eclipse Escet.

### 3.3. Імітаційне моделювання реалізації супервайзингу робота

Основна ідея мови полягає в тому, що код генерується зі скрипту DSL. Цей код можна розгорнути в проміжному програмному забезпеченні для роботів, і він одразу ж запуститься. Це означає, що користувачі можуть використовувати засоби налагодження, що надаються згенерованою мовою (наприклад, C++ для ROS2). Але це може бути громіздким для налаштування та розуміння. Також дуже ймовірно, що користувачі мало або зовсім не знають цих згенерованих мов.

Щоб спростити налагодження поведінки контролера, було додано налаштовувану опцію, яка визначає, чи повинен вузол контролера публікувати інформацію про поточний стан усіх компонентів у тему. Інші вузли можуть підписатися на цю тему та обробляти інформацію.

Дані, що публікуються в тему, є серіалізованим рядком JSON. Хоча можна було б використовувати власне визначення інтерфейсу, JSON було обрано, оскільки він не вимагає додаткових залежностей на цільовій платформі. Поточний стан серіалізується в JSON і містить таку інформацію:

- Компоненти
- Стани компонентів
- Переходи компонентів
- Змінні компонентів
- Значення змінних
- Виконані переходи

Хоча серіалізована інформація вже надає деяку інформацію, її все ще важко читати. Ось чому розширення Visual Studio Code має функцію, яка

візуалізує інформацію. Розширення запускає вузол у фоновому режимі та підписується на тему стану контролера. Як тільки надходить інформація, воно або створює візуальне представлення автоматів та змінних, або оновлює існуючу візуалізацію. Якщо надходять дані від нового робота і, таким чином, змінюється визначення робота, воно створить нове візуальне представлення. Розширення було створено за допомогою JavaScript та використовує пакети ROS від Foxglove для полегшення зв'язку з проміжним програмним забезпеченням та Dagre D3 для візуалізації кінцевих автоматів. Приклад такої візуалізації можна знайти на рисунку 3.5.

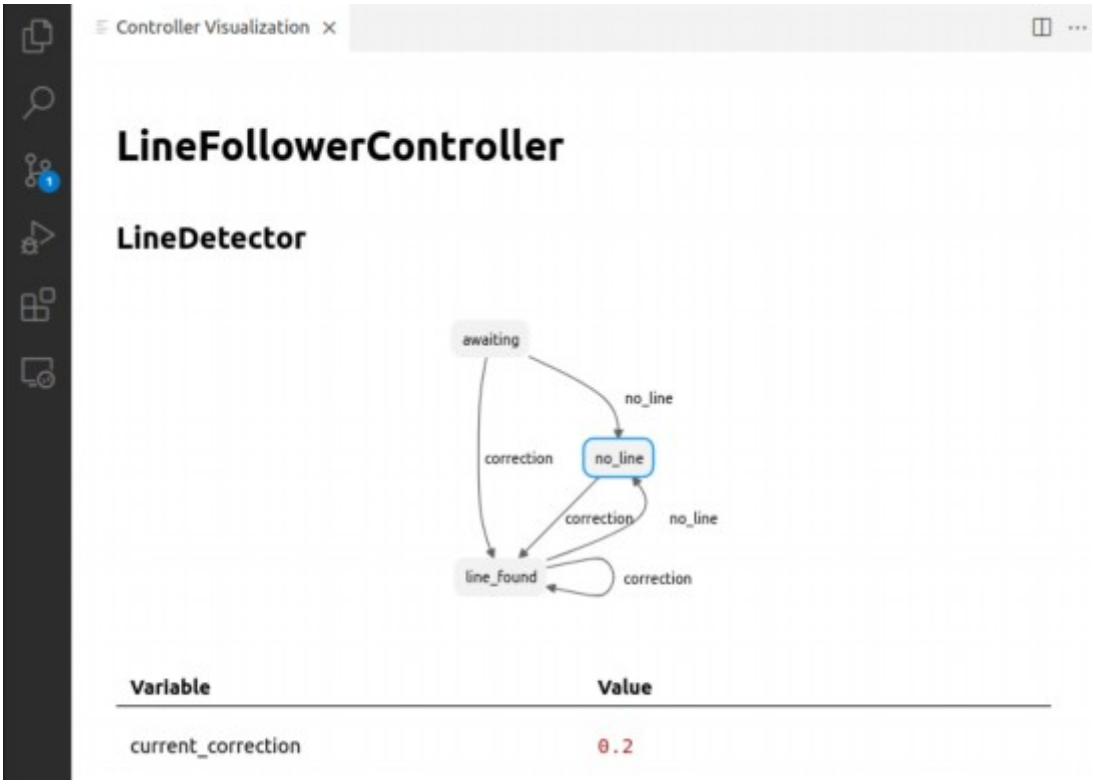


Рис. 3.5. Приклад візуалізації стану робота у Visual Studio Code

Також можна зібрати більше інформації про вхід та вихід контролера за допомогою текстового файлу. За замовчуванням ведення журналу не генерується. Якщо користувач налаштовує ведення журналу, вузол контролера записує всі події у файл разом із позначкою часу, яку можна використовувати для аналізу поведінки, яка виконується або була виконана вузлом контролера.

Розглянемо сценарій де використовується супервізор. Він використовує контролер, отриманий зі сценарію стеження за лінією, але без аварійної зупинки та датчика Lidar. Аварійна зупинка та датчик LiDAR застосовуються за допомогою супервізора. Уся комунікація з контролером і до нього проходить через супервізор. Якщо аварійну зупинку активовано, будь-який рух блокується супервізором.

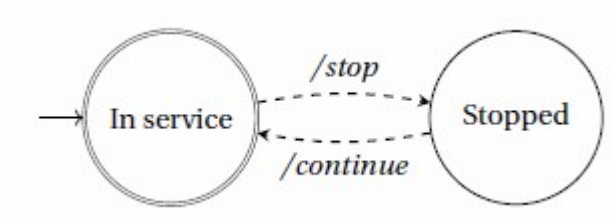


Рис. 3.6. Стани компонента аварійної зупинки

Для сценарію розроблено контролер з використанням мови, описаної в цій роботі. Рух у цьому сценарії трохи відрізняється від сценарію стеження за лінією. Стеження за лінією публікувало кутову або лінійну швидкість у тему, що змушувало робота рухатися таким чином, доки не буде отримано нову швидкість.

Щоб дозволити використання супервізора, для цілей цієї дисертації було розроблено вузол, який може приймати кроки руху. Він підтримує ті самі вхідні дані, що і звичайна команда TurtleBot, але він виконуватиме її лише протягом 100 мс, після чого робот знову зупиниться.

Таким чином, супервізор може перехоплювати команди руху та перевіряти, чи дозволяє він їх. Якщо ні, він може відхилити команду, що станеться у випадку, якщо натиснуто аварійну зупинку або щось знаходиться перед роботом.

Модель для супервізора дуже схожа на контролер стеження за лінією і додана в лістингу 3.6.

Комунікація, як видно з компонента, показана в таблиці 3.2.

Таблиця 3.2.

Огляд комунікації сценарію з використанням супервізора

| Direction | Type | Name | Data Type | Component |
|---|---|---|---|---|
| Outgoing | Message | /correction | double | Line Detector |
| Outgoing | Message | /correction | double | Line Detector |
| Outgoing | Message | /no_line | none | Line Detector |
| Outgoing | Message | /stop | none | Emergency Stop |
| Outgoing | Message | /continue | none | Emergency Stop |
| Incoming | Message | /simple_movement | Twist | Simple Movement |

Лістинг 3.6. Код DSL для сценарію супервізора

```
robot LineFollowerSupervised {
    interface laser use LaserScan from sensor_msgs
    interface movement use Twist from geometry_msgs

    datatype object Vector3 {
        x: double
        y: double
        z: double
    }

    datatype object Twist {
        linear: Vector3
        angular: Vector3
    }

    component LineDetector {
        outgoing message correction with identifier: "/correction", type: double
        outgoing message no_line with identifier: "/no_line", type: none

        behaviour {
            variable current_correction: double = 0.0

            initial marked state no_line {
                on response from correction goto line_found
            }

            state line_found {
                on response from no_line goto no_line
                on response from correction do current_correction := value
            }
        }
    }

    datatype object LaserScan {
        ranges: array(double)
    }

    datatype enum DistanceSafety from LaserScan to {
        value.ranges[0] > 1.0 and value.ranges[270] > 0.5 and value.ranges[90] > 0.5 and value
            .ranges[45] > 0.7 and value.ranges[305] > 0.7—> safe
        default —> unsafe
    }

    component LidarSensor {
        outgoing message scan with identifier: "/scan", type: DistanceSafety links laser

        behaviour {
            variable current_distance: DistanceSafety = unsafe
```

```
            on response from scan do current_distance := value

            initial marked state unsafe_distance {
                transition if current_distance = safe goto safe_distance
            }

            state safe_distance {
                transition if current_distance = unsafe goto unsafe_distance
            }
        }
    }

    component SimpleMovement {
        incoming message move with identifier: "/simple_movement", type: Twist links movement
    }

    component EmergencyStop from EmergencyStopLibrary import EmergencyStop

    requirement move needs LineDetector.line_found
    requirement move needs EmergencyStop.in_service
    requirement move needs LidarSensor.safe_distance

    provide move with {
        linear: { x: 0.6 },
        angular: { z: (-LineDetector.current_correction) / 100 }
    }
}
```

Сценарій був змодельований за допомогою інструменту моделювання Gazebo. Gazebo - це симулятор з відкритим вихідним кодом для перевірки поведінки робота за допомогою симулятора. Хоча симулятор Gazebo за замовчуванням можна використовувати без ROS, існують пакети, що дозволяють інтеграцію між ROS та Gazebo [41].

Сценарій вимагає запуску кількох вузлів. Для підтримки цього кожен з вузлів був оснащений пакетом сценарію, який містить файл запуску, який запускатиме симуляцію Gazebo, всі вузли сценарію, аварійну зупинку та будь-які додаткові необхідні інструменти, такі як Rviz.

Для порівняння було змодельовано два сценарії, з використанням та без використання супервайзора.

У першому сценарії (рис. 3.7), стеження за лінією, можна спостерігати, що стеження за лінією слідує за жовтою лінією, доки не перестане знаходити лінію. Якщо між ними натиснути аварійну зупинку, робот припиняє рух. Те саме відбувається, коли об'єкт розміщується перед роботом. Це тимчасово зупинить процес виявлення лінії. Він трохи відхиляється від лінії, коли зупиняється, оскільки йому потрібно зупинити всю кутову швидкість.

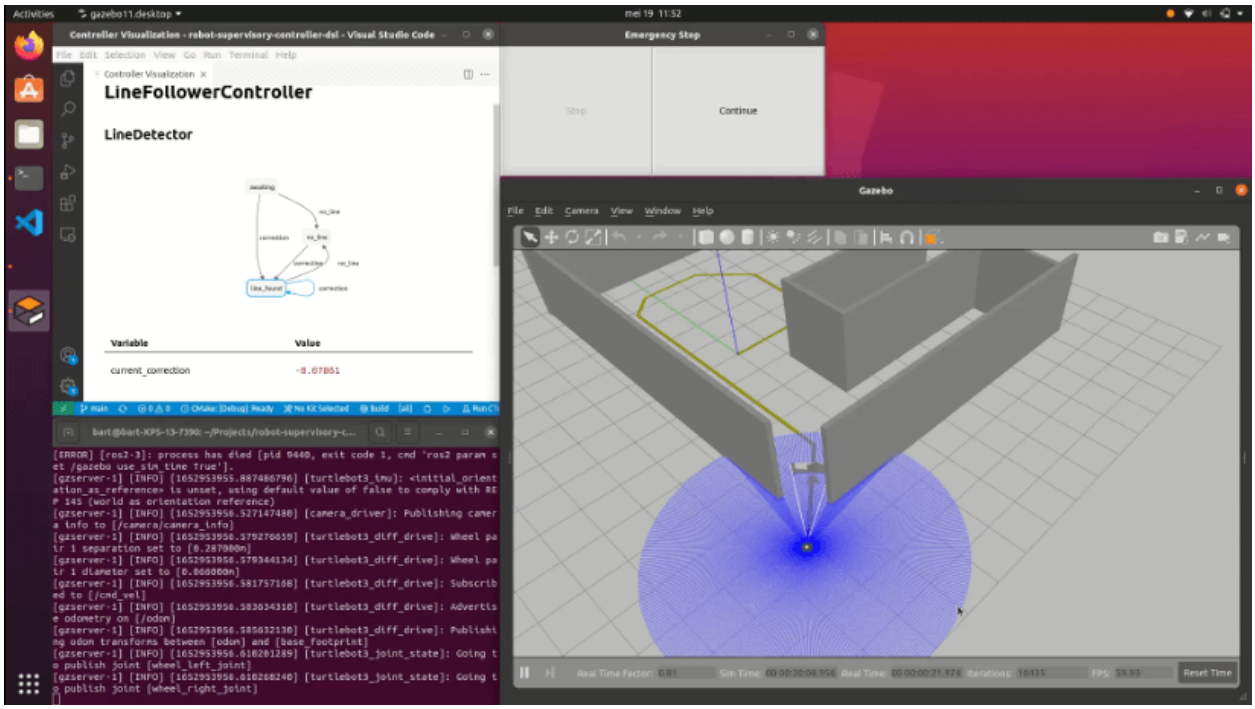


Рис. 3.7. Моделювання сценарію стеження за лінією без супервайзора

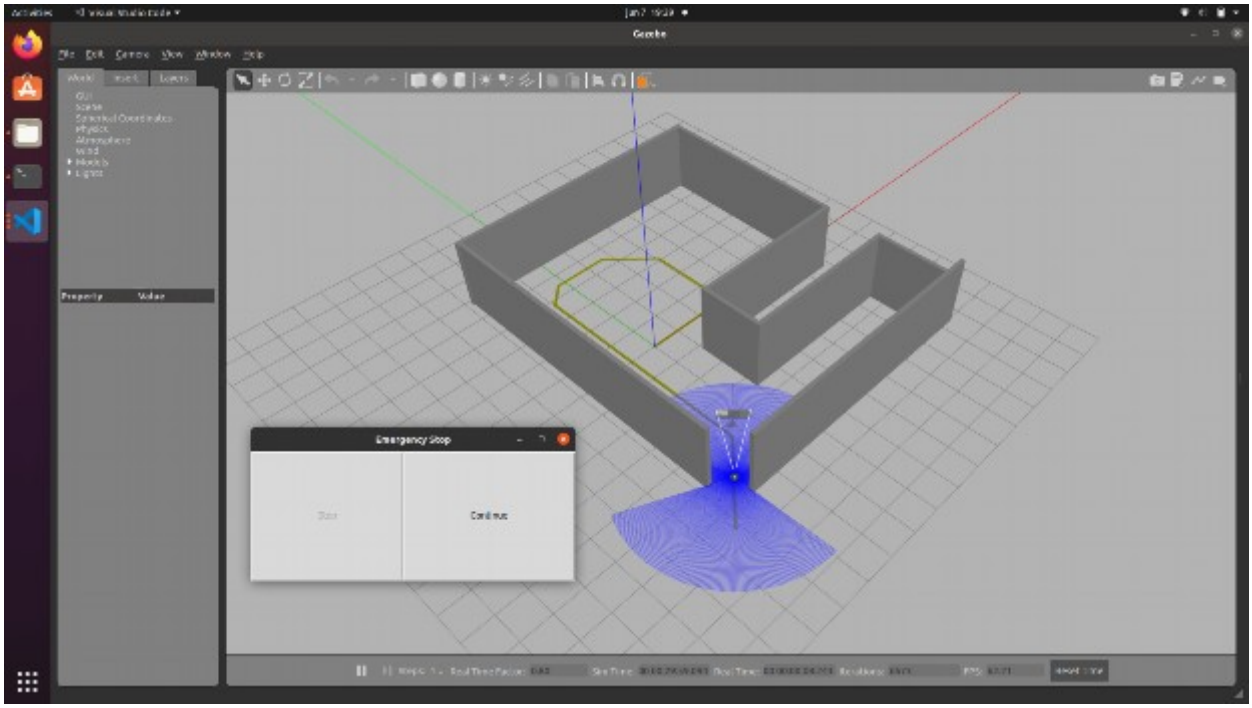


Рис. 3.8. Моделювання сценарію стеження за лінією з супервайзором

Другий сценарій (рис. 3.8) демонструє поведінку, подібну до першого сценарію, оскільки обидва реалізують робота, що стежить за лінією. У цьому випадку це реалізовано за допомогою супервізора, який вимагав додаткового

вузла, що дозволяв рух робота в кроках часу. Найважливіша частина супервізора полягає в тому, що він зупиняє робота, коли натиснуто аварійну зупинку. Зверніть увагу, що коли сценарій запущено, запускаються як контролер, так і супервізор, і що натискання аварійної зупинки коректно зупиняє робота.

## Висновки до розділу

Третій розділ присвячений імплементації методів та моделей доменно-специфікованих мов (DSL) для супервайзингу, що дозволяє генерувати адаптивні та функціональні аплікації. Запропоновано концепцію, яка забезпечує автоматизовану генерацію артефактів, таких як код, конфігураційні файли чи моделі, необхідні для реалізації супервайзингових функцій. Це дозволяє спростити та прискорити процес розробки робототехнічних систем. Виконано імітаційне моделювання для перевірки працездатності запропонованих підходів. Результати моделювання підтвердили ефективність застосування DSL для проектування супервайзингових систем та поліпшення продуктивності та надійності роботи робототехнічних систем завдяки автоматизованому аналізу станів і адаптації до змін середовища.

# ВИСНОВКИ

В магістерській роботі проведено комплексне дослідження застосування доменно-специфікованих мов (DSL) у контексті супервайзингового управління робототехнічними системами, що включало аналіз предметної області, розробку моделей і методів, а також їхню практичну імплементацію.

Визначено особливості модельно-керованої інженерії (MDE) як підходу, що дозволяє зменшити розрив між концептуальною моделлю системи та її реалізацією завдяки автоматизованим трансформаціям.

Проведено аналіз доменно-специфікованих мов, що використовуються в робототехніці, та обґрунтовано їхню ефективність для формалізації процесів управління. Розглянуто теорію супервайзингового контролю як основу для створення систем управління, що гарантують коректність і надійність функціонування робототехнічних систем.

Запропоновано підхід до створення предметно-орієнтованих мов для робототехніки, що враховує специфіку роботи систем у реальному часі. Розроблено критерії оцінки DSL, зокрема зручність використання, продуктивність, масштабованість та інтеграція в екосистему робототехнічних систем. Деталізовано процеси комунікації, управління компонентами та обробки даних у контексті супервайзингового управління, що дозволяє забезпечити узгодженість і адаптивність роботи робототехнічних систем.

Реалізовано концепцію генерації артефактів на основі DSL, що спрощує автоматизацію розробки систем управління. Інтегровано теорію супервайзингового контролю в процеси управління робототехнічними системами через DSL, забезпечивши формалізацію правил поведінки та коректності. Виконано імітаційне моделювання для перевірки запропонованих рішень, яке підтвердило ефективність і придатність розроблених моделей для реальних умов.

Результати роботи свідчать про те, що використання доменно-специфікованих мов у поєднанні з модельно-керованою інженерією та теорією супервайзингового контролю є перспективним напрямом у розробці робототехнічних систем. Це дозволяє підвищити їхню функціональність, надійність і адаптивність до змін середовища, що відкриває нові можливості для створення складних інтерактивних систем управління.

В роботі продемонстровано практичну реалізацію методів та моделей DSL для супервайзингу, які дозволяють створювати гнучкі та адаптивні робототехнічні системи. Результати імітаційного моделювання підтверджують ефективність розроблених рішень та їхню придатність до використання в реальних застосуваннях.

# ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ


1. Maurice H ter Beek, Michel A Reniers, and Erik P de Vink. Supervisory controller synthesis for product lines using cif 3. In International Symposium on Leveraging Applications of FormalMethods, pages 856–873. Springer, 2016.
2. AbhishekBhattacharjee andDaniel Lustig. Architectural and operating system support for virtual memory. Synthesis Lectures on Computer Architecture, 12(5):1–175, 2017.
3. Hendrik Bünder. Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages. In MODELSWARD, pages 129–140, 2019.
4. Jianping Cai, Xuting Wan, Meimei Huo, and Jianzhong Wu. An algorithm of micromouse maze solving. In 2010 10th IEEE International Conference on Computer and Information Technology, pages 1995–2000. IEEE, 2010.
5. Sukting Chong, Joe Dinius, and dps53. Turtlebot: Turtlebot 3 waffle pi. https://www.robotis.us/turtlebot-3-waffle-pi/.
6. Anne-Lise Courbis, Kahune Luu, Benjamin Grondin, and Kelly Roussel. A model driven architecture framework for robot design and automatic code generation. In 15th China-Europe International Symposium on Software Engineering Education, 2019.
7. Fowler, M. (2010). Domain-Specific Languages. Addison-Wesley.
8. Van Deursen, A., Klint, P., & Visser, J. (2000). Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices, 35(6), 26-36.
9. Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. ACM Computing Surveys, 37(4), 316-344.
10. Hudak, P. (1998). Modular domain-specific languages and tools. Proceedings of the 5th International Conference on Software Reuse. IEEE.
11. Erdweg, S., et al. (2015). The State of the Art in Language Workbenches. Software and Systems Modeling, 14(2), 571-604.

12. Spinellis, D. (2001). Notable design patterns for domain-specific languages. Journal of Systems and Software, 56(1), 91-99.
13. Deursen, A., & Klint, P. (1998). Little languages: Little maintenance?. Journal of Software Maintenance, 10(2), 75-92.
14. Visser, E. (2004). Program transformation with Stratego/XT. ACM SIGPLAN Notices, 39(11), 168-177.
15. Kosar, T., Bohra, Z., & Mernik, M. (2016). Domain-Specific Languages: A Systematic Mapping Study. Information and Software Technology, 71, 77-91.
16. Antkiewicz, M., & Czarnecki, K. (2006). Framework-Specific Modeling Languages with Round-Trip Engineering. MoDELS 2006. Springer.
17. Kelly, S., & Tolvanen, J. P. (2008). Domain-Specific Modeling: Enabling Full Code Generation. Wiley.
18. Wile, D. S. (2001). Supporting the DSL spectrum. Journal of Computing and Information Technology, 9(4), 263-287.
19. Czarnecki, K., & Helsen, S. (2003). Classification of Model Transformation Approaches. Proceedings of the 2nd OOPSLA Workshop on Generative Techniques.
20. Bravenboer, M., & Visser, E. (2004). Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. ACM SIGPLAN Notices, 39(10), 365-383.
21. Schätz, B., et al. (2010). DSLs for the software development lifecycle. ACM SIGSOFT Software Engineering Notes, 35(5), 3-11.
22. Brand, M. G. J. (2001). Code generation from DSLs: A tutorial. ACM Transactions on Software Engineering and Methodology, 10(4), 384-422.
23. Oliveira, B. C. D. S., & Cook, W. R. (2012). Extensibility for the Masses. ACM Transactions on Programming Languages and Systems, 33(2), 1-30.
24. Karsai, G., et al. (2003). Model-Integrated Development of Cyber-Physical Systems. IEEE Proceedings of the IEEE, 91(1), 27-40.

25. Gray, J., & Rossi, M. (2004). Visual DSL Design in Software Product Lines. Software Product Line Conference. IEEE.
26. Lämmel, R., & Visser, J. (2003). Declarative XML Processing with Haskell. Journal of Functional Programming, 13(5), 485-508.
27. Batory, D. S. (2004). Feature-oriented programming and the AHEAD tool suite. Proceedings of the 26th International Conference on Software Engineering.
28. Rahm, E., & Do, H. H. (2001). Data Cleaning: Problems and Current Approaches. IEEE Data Engineering Bulletin, 23(4), 3-13.
29. Schwarzmüller, M. (2020). The Modern JavaScript Bootcamp. O'Reilly.
30. Hudak, P., & Jones, M. P. (1994). Haskell as a DSL for Algorithm Design. Journal of Functional Programming, 3(4), 335-354.
31. Pnueli, A., & Shalev, M. (1991). What is in a Step: On the Semantics of Statecharts. ACM SIGSOFT Software Engineering Notes, 16(3), 158-169.
32. Schmidt, D. C. (2006). Model-Driven Engineering. IEEE Computer, 39(2), 25-31.
33. Fowler, M., & Parsons, R. (2011). DSLs and Design Patterns. Addison-Wesley.
34. Lanz, A., et al. (2020). Executable Domain-Specific Modeling Languages: A Case Study. ACM Transactions on Software Engineering and Methodology, 29(3), 1-41.
35. Cook, W. R. (2012). On Understanding Data Abstraction, Revisited. ACM SIGPLAN Notices, 47(12), 557-572.
36. Dijkstra, E. W. (1974). Programming Considered as a Human Activity. ACM Computing Surveys, 6(4), 357-372.
37. Stahl, T., & Völter, M. (2006). Model-Driven Software Development: Technology, Engineering, Management. Wiley.
38. Mellor, S. J., et al. (2004). MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley.

39. Kleppe, A., Warmer, J., & Bast, W. (2003). MDA Explained: The Model Driven Architecture. Addison-Wesley.
40. Zaytsev, V. (2012). Grammar-Based Testing for Domain-Specific Languages. ACM SIGPLAN Notices, 47(11), 1-18.
41. Ghosh, D. (2010). DSLs in Action. Manning.
42. Heering, J., et al. (1989). Realistic Compilation by Partial Evaluation. ACM Transactions on Programming Languages and Systems, 11(3), 321-365.
43. Tarlecki, A. (1997). Specifications for Domain-Specific Models. Journal of Systems and Software, 38(1), 121-143.
44. Stefik, A., & Siebert, S. (2013). The Influence of Syntax on Novices in Programming. ACM SIGCSE Bulletin, 41(3), 211-215.
45. Magnusson, B., & Hedin, G. (2009). Generating Interpreters for DSLs. ACM SIGPLAN Notices, 44(10), 125-132.
46. Metayer, D. L. (1996). Syntax and Semantics of DSLs: A Formal Perspective. ACM Transactions on Software Engineering and Methodology, 5(4), 345-364.
47. Alberto Rodrigues Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems & Structures, 43:139–155, 2015.
48. In 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 8308–8314. IEEE, 2018.
49. Nico Hochgeschwender and Sebastian Wrede. Dsls in robotics: A case study in programming self-reconfigurable robots. Grand Timely Topics in Software Engineering: International Summer School GTTSE 2015, Braga, Portugal, August 23-29, 2015, Tutorial Lectures, 10223:98, 2017.
50. NicoleHutchins. A dsml for a robotics environment to support synergistic learning of ct and geometry. kong, sc, sheldon, j., & li, ky.(eds.). conference. In Proceedings of International Conference on Computational Thinking Education 2018.