БАКАЛАВРСЬКА РОБОТА

БР. ІП - 83.00.00.000 ПЗ

Група ІП-22-3

Сорохман Богдан

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Сорохман Богдан Петрович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Розробка програмного забезпечення кіберфізичних систем
(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

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

Здобувач освітнього ступеня _____Сорохман Б.П._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____Крихівський Михайло Васильович, к.т.н., доцент
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Сорохман Богдану Петровичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Розробка програмного забезпечення кіберфізичних систем"

керівник проекту (роботи) Крихівський Михайло Васильович, к.т.н., доцент

затверджені наказом вищого навчального закладу від “ 21 ” квітня 2026 р. № 179/7

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

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

1. Аналіз вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

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

1 Огляд кіберфізичної системи, що використовується в цій роботі. (рис. 1.1, ст. 14)
2 Проста модель на основі елементів Component Port Connection (рис. 14, ст. 29)
3 Модель архітектури для прикладу кіберфізичної системи (рис. 2.4, ст. 46)
4 Огляд необхідних кроків для перетворення моделі на виконуваний за стосунок (рис. 2.7, ст. 52)
5 Діаграма кінцевого автомата життєвого циклу GAC (рис. 3.3, ст. 69)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	17.01.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.02.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.03.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.04.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	02.05.2026	виконано

Студент _____Сорохман Б.П.
(підпис)

Керівник роботи _____Крихівський М.В.
(підпис)

АНОТАЦІЯ

Бакалаврська робота робота містить 98 сторінки, 25 рисунків, список використаних джерел із 40 найменувань.

Метою роботи є дослідження принципів та підходів до розробки програмного забезпечення кіберфізичних систем, а також аналіз методів побудови надійних, масштабованих і ефективних систем реального часу.

Об'єкт дослідження: кіберфізичні системи та процеси їх програмної реалізації.

Предмет дослідження: методи, моделі та інструменти розробки програмного забезпечення кіберфізичних систем, зокрема вбудованих систем керування, модельно-орієнтованого проєктування та систем реального часу.

Результати дослідження: проведено аналіз архітектури кіберфізичних систем, розглянуто методи розробки та реалізовано програмне забезпечення кіберфізичної системи з подальшою оцінкою її ефективності.

У першому розділі розглянуто теоретичні основи кіберфізичних систем, принципи їх функціонування, особливості забезпечення роботи в режимі реального часу та модельно-орієнтований підхід до розробки ПЗ.

У другому розділі досліджено методи та інструменти розробки програмного забезпечення для кіберфізичних систем, включаючи підходи до проєктування вбудованих систем керування та моделювання архітектури.

У третьому розділі представлено проєктування загальної архітектури кіберфізичної системи, реалізацію програмного забезпечення та аналіз ефективності функціонування системи.

Висновок: використання сучасних методів розробки програмного забезпечення для кіберфізичних систем дозволяє забезпечити високу надійність, адаптивність та ефективність роботи систем реального часу.

КЛЮЧОВІ СЛОВА: КІБЕРФІЗИЧНІ СИСТЕМИ, ВБУДОВАНІ СИСТЕМИ, СИСТЕМИ РЕАЛЬНОГО ЧАСУ, АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, АВТОМАТИЗАЦІЯ, IoT, ВЕРИФІКАЦІЯ.

ANNOTATION

Bachelor's thesis of 98 pages, 25 figures, a reference list containing 40 sources.

The aim of the work is to study the principles and approaches to software development for cyber-physical systems and to analyze methods for building reliable, scalable, and efficient real-time systems.

Research object: cyber-physical systems and the processes of their software implementation.

Research subject: methods, models, and tools for software development of cyber-physical systems, including embedded control systems, model-based design, and real-time systems.

Research results: the architecture of cyber-physical systems was analyzed, the principles of real-time systems were studied and software for a cyber-physical system was implemented and evaluated.

The first section describes the theoretical foundations of cyber-physical systems, principles of their operation, real-time guarantees, and the model-based approach to software development.

The second section analyzes methods and tools for software development of cyber-physical systems, including approaches to embedded control system design, software architecture modeling, and testing and verification tools.

The third section presents the design of the overall architecture of a cyber-physical system, software implementation, and analysis of system performance and efficiency.

Conclusion: the use of modern software development methods for cyber-physical systems ensures high reliability, adaptability, and efficiency of real-time systems, as well as improves automation and safety of technological processes.

KEY WORDS: CYBER-PHYSICAL SYSTEMS, EMBEDDED SYSTEMS, REAL-TIME SYSTEMS, MODEL-BASED DEVELOPMENT, SOFTWARE ARCHITECTURE, AUTOMATION, IoT, VERIFICATION.

ЗМІСТ

					БР.ІП –83.00.00.000 ПЗ		
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*			
Розроб.		Сорохман Б.П.			Розробка програмного забезпечення кіберфізичних систем	*Літ.*	*Арк.* 7 / *Акрушів*
Перевір.		Крихівський М.В.					
Реценз.		Процюк В.Р.				ІФНТУНГ ІП-22-3	
Н. Контр.		Піх М.М.			**Пояснювальна записка**		
Затверд.		Бандура В. В.					

ВСТУП

У сучасному світі кіберфізичні системи стають важливою складовою цифрової трансформації промисловості, транспорту, енергетики, медицини та автоматизованих виробничих процесів. Активний розвиток технологій Інтернету речей, вбудованих систем, сенсорних мереж і систем реального часу сприяє інтеграції фізичних процесів із програмними компонентами, що дозволяє створювати інтелектуальні системи керування та моніторингу. Сучасні виклики, пов’язані з необхідністю обробки великих обсягів даних у режимі реального часу, забезпеченням високої надійності та безпеки функціонування систем, підкреслюють важливість розробки ефективного програмного забезпечення для кіберфізичних систем.

Актуальність теми зумовлена стрімким розвитком автоматизованих систем керування та необхідністю створення програмного забезпечення, здатного забезпечувати взаємодію між фізичними пристроями, сенсорами, виконавчими механізмами та програмними компонентами в єдиному інформаційному середовищі. Існуючі рішення часто мають обмеження щодо масштабованості, продуктивності та підтримки роботи в режимі реального часу. Крім того, складність інтеграції апаратного та програмного забезпечення потребує використання сучасних підходів до моделювання, проєктування та тестування систем. У контексті розвитку концепцій Industry 4.0 та Smart Systems створення ефективних програмних рішень для кіберфізичних систем є важливим напрямом розвитку сучасних інформаційних технологій.

Метою роботи є дослідження принципів та методів розробки програмного забезпечення кіберфізичних систем, а також проєктування та реалізація програмної системи, що забезпечує надійне функціонування компонентів у режимі реального часу.

Завданнями дослідження є аналіз предметної області та існуючих підходів до побудови кіберфізичних систем, визначення функціональних і нефункціональних вимог до програмного забезпечення, дослідження методів забезпечення роботи в режимі реального часу, аналіз модельно-орієнтованих підходів до

розробки, проєктування архітектури системи, реалізація програмних компонентів, тестування та оцінка ефективності функціонування системи. У процесі дослідження буде обґрунтовано необхідність використання сучасних методів верифікації та моделювання для підвищення надійності й продуктивності кіберфізичних систем.

Об'єктом дослідження є процеси розробки програмного забезпечення кіберфізичних систем, що включають проєктування архітектури, реалізацію, інтеграцію апаратних і програмних компонентів, тестування та забезпечення функціонування систем у режимі реального часу.

Предметом дослідження є методи, технології та інструменти розробки програмного забезпечення кіберфізичних систем, зокрема модельно-орієнтована розробка, архітектурне моделювання, вбудовані системи керування, засоби тестування та верифікації програмного забезпечення.

Методи дослідження включають аналіз наукових джерел для вивчення теоретичних основ кіберфізичних систем, порівняльний аналіз технологій і підходів до проєктування, моделювання архітектури системи, експериментальний метод для реалізації та тестування програмного забезпечення, а також аналіз результатів функціонування системи.

Розроблене програмне забезпечення передбачатиме інтеграцію компонентів кіберфізичної системи, підтримку роботи в режимі реального часу, взаємодію з сенсорами та виконавчими пристроями, а також можливість масштабування та адаптації до різних сфер застосування. Особлива увага приділятиметься надійності, продуктивності та безпеці системи, що дозволить забезпечити стабільну роботу в умовах високих навантажень та критично важливих процесів. Очікується, що результати роботи сприятимуть підвищенню ефективності автоматизованих систем керування та розвитку сучасних кіберфізичних технологій.

Бакалаврська робота містить 98 сторінки, 25 рисунків, 3 розділи, список використаних джерел із 40 найменувань.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ КІБЕРФІЗИЧНИХ СИСТЕМ

1.1 Основи кіберфізичних систем та принципи їх функціонування

Розробка програмного забезпечення для керування сучасними кіберфізичними системами стає дедалі складнішою через зростання кількості та складності їхніх вимог, як зазначено в дорожній карті RoboNED з робототехніки. Механічні конструкції фізичних частин систем стають дедалі складнішими. В результаті системи мають дедалі більшу кількість датчиків та виконавчих механізмів, які необхідні програмному забезпеченню керування для забезпечення бажаної поведінки системи. Для систем загального призначення гнучкість та складність їхніх поведінкових завдань також зростають, щоб зробити систему придатною для якомога більшої кількості ситуацій. Незважаючи на те, що складність продовжує зростати, системи повинні ставати дедалі безпечнішими. Зростаюча потреба в безпеці виникає через зростаючу взаємодію з людьми та іншими кіберфізичними системами [1]. Крім того, кіберфізичні системи також прагнуть стати більш мобільними, а отже, повинні бути максимально енергоефективними.

На відміну від цих вимог до якості, таких як безпека, надійність та енергоефективність, компанії, які розробляють та виробляють кіберфізичні системи, мають вимогу щодо часу виведення продукту на ринок. Вони воліють не витрачати забагато часу на вимоги до якості, оскільки це уповільнить процес проектування та, таким чином, відкладе доступність їхнього продукту на ринку. Це невигідно для компанії, коли є конкуренти, які розробляють аналогічний продукт, або просто тому, що надходження доходів від продукту також затримуються.

Типовим прикладом сучасних кіберфізичних систем, що відповідають вищезазначеним вимогам, є медичні роботизовані системи. Традиційні медичні системи вже добре зарекомендували себе в лікарнях, наприклад, у вигляді систем візуалізації, таких як МРТ або ультразвукові сканери. Окрім цих традиційних систем, у лікарнях впроваджується новий тип роботизованих систем [2]. Ці нові системи мають більш інтенсивну взаємодію з людьми в лікарнях, як з

персоналом, так і з пацієнтами. Прикладами є роботизовані системи , які допомагають піднімати пацієнтів з ліжка або класти їх у ліжка, автоматизовані візки, що перевозять їжу або білизну по лікарні, або навіть роботизовані системи, які безпосередньо беруть участь у проведенні хірургічних втручань на пацієнті.

Прикладами систем, що безпосередньо задіяні в роботизованій хірургії. Ця дипломна робота фінансується проектом TeleFLEX. Метою цього проекту є дослідження, проектування та створення роботизованого ендоскопа, який можна використовувати для проведення біопсій та простих операцій на кишечнику людини. Ці роботизовані системи спрямовані на допомогу в загальних операціях, використовуючи мінімально інвазивний підхід, що призводить до меншої травматизації для пацієнта. Сучасні кіберфізичні системи додатково підтримують можливість перебування хірурга в іншому місці, тому система керується дистанційно за допомогою Інтернету або будь-якої іншої мережі. Окрім покращення переваг для пацієнтів, деякі з цих нових систем також покращують переваги особисто для хірургів , наприклад, проект TeleFLEX спрямований на покращення робочої пози хірургів.

Описані медичні кіберфізичні системи є складними, мають значну взаємодію з оператором та його середовищем, і тому вимагають великої кількості датчиків та виконавчих механізмів. Програмне забезпечення керування повинно обробляти велику кількість потоків даних та зв'язку, забезпечуючи, щоб усі частини програмного забезпечення могли своєчасно розраховувати свої алгоритми керування [4]. Аспекти надійності та безпеки необхідні для того, щоб бути абсолютно впевненим у тому, що система поводиться належним чином у всіх ситуаціях, інакше система може завдати шкоди пацієнту з потенційно катастрофічними наслідками.

Вище показано, що сучасні кіберфізичні системи стають дедалі складнішими з різних причин, в результаті чого програмне забезпечення керування також стає складнішим. Крім того, апаратне забезпечення, яке виконує програмне забезпечення, також стає більш потужним та багатофункціональним [3]. Наприклад, сучасне апаратне забезпечення часто має процесор з кількома ядрами.

Програмне забезпечення, яке повинно ефективно використовувати їх, є складнішим, ніж одноядерна реалізація. Ця зростаюча складність програмного забезпечення вимагає нових методів проектування для ефективного управління цією зростаючою складністю.

Огляд кіберфізичної системи

Терміни *кіберфізична система* та *роботизована система* використовувалися в попередньому розділі. Крім того, термін *мехатронна система* також є добре відомим. Ці три терміни використовуються для опису одного типу системи, тобто системи, яка складається з механічної частини, електронної частини та програмної (керуючої) частини.

- Термін «мехатронна система» більше зосереджений на мехатроніці.
- Термін «робототехнічна система» справді зосереджується на робототехнічному застосуванні.
- Термін «кіберфізична система» справді зосереджується на наявності кібер- та фізичної частини.

Автор вирішив використовувати переважно термін *«кіберфізична система»,* оскільки він зосереджений на наявності кіберчастини системи, що є основною темою цієї роботи. Однак у певних ситуаціях термін «роботизована система» є більш застосовним, тому він використовується в цих ситуаціях. Фактичні компоненти цих систем пояснюються в решті цього розділу.

Кіберфізична система складається з *кібердомену* та *фізичного домену* , як показано на рисунку 1.1. Фізичний домен зазвичай складається з механічних та електричних компонентів системи. Виконавчі механізми та датчики перетворюють сигнали між *електричним* та механічним *доменами* . Кібердомен складається з програмного забезпечення, яке керує системою, і воно знаходиться на (вбудованій) *обчислювальній платформі.* Електричні сигнали перетворюються цифро-аналоговими перетворювачами (ЦАП) та аналого-цифровими перетворювачами (АЦП) для зв'язку з кібердоменом та його програмними компонентами та з ним відповідно.

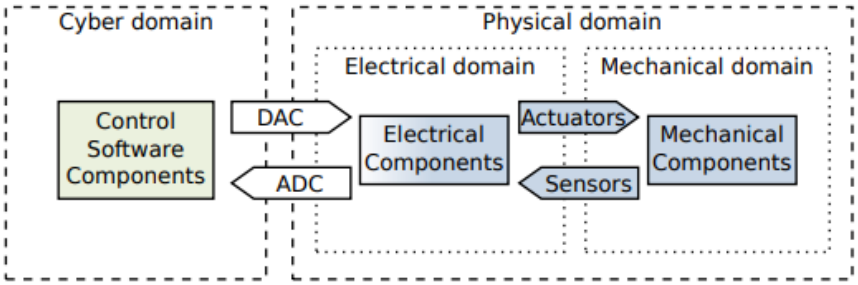

Рисунок 1.1 - Огляд кіберфізичної системи, що використовується в цій роботі.

Існує кілька програмних рішень для керування кіберфізичною системою. Агентні контролери використовують інтелектуальні, автономні сутності (агенти), які взаємодіють із середовищем для роботи над певним завданням. Багатоагентні системи (MAS) використовують кількох агентів для виконання складніших завдань.

Інший спосіб структурування програмного забезпечення для керування полягає в реалізації архітектури субсумпції. Така архітектура складається з кількох модулів, організованих шарами один над одним, кожен з яких реалізує свою конкретну мету [5]. Модулі та шари здатні пригнічувати або блокувати виходи своїх вищих шарів.

Багаторівневий підхід, що використовується в цій роботі, зображено на рисунку 1.2. Кожен шар має свій власний набір властивостей, які впливають на поведінку компонентів керування, розміщених на них. Компоненти або безпосередньо керують кіберфізичною системою, або надають завдання іншим компонентам впливати на загальну поведінку кіберфізичної системи. Оскільки ця робота зосереджена на кібердомені кіберфізичних систем, частина фізичного домену спрощується блоками « *Установка* » та частиною *апаратного забезпечення вводу/виводу* . Кібердомен, або *вбудоване програмне забезпечення керування,* показано більш детально ліворуч на рисунку. Подальші деталі компонента вбудованого програмного забезпечення керування, показаного на рисунку 1.2.

Типовим використанням архітектури є поєднання вбудованої обчислювальної платформи з FPGA. FPGA використовується для отримання доступу до багатьох легко використовуваних контактів вводу/виводу, які використовуються

для підключення до виконавчих механізмів та датчиків. Програмний драйвер обробляє зв'язок з FPGA, тому програмне забезпечення керування має прямий доступ до контактів FPGA і, таким чином, до сигналів виконавчих механізмів та датчиків.

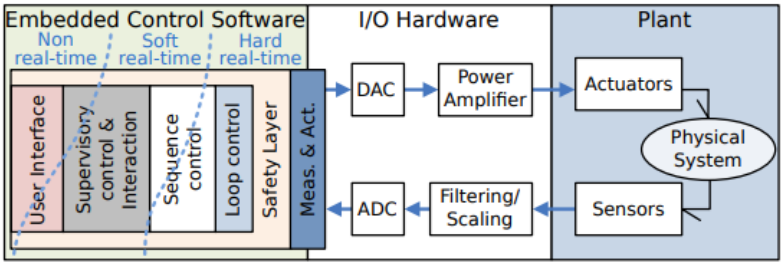

Рисунок 1.2 - Детальна системна архітектура кіберфізичної системи

Як згадувалося раніше, ця робота зосереджена на проектуванні та впровадженні вбудованого програмного забезпечення для керування [6]. Однак під час проектування та впровадження програмного забезпечення необхідно враховувати й інші області, оскільки знання цих областей необхідні під час розробки програмного забезпечення для отримання оптимального результату.

Ця робота зосереджена на трьох основних цілях. Ці цілі призводять до покращення якості програмного забезпечення для керування кіберфізичними системами та до швидшої та більш структурованої траєкторії проектування.

- Основна мета цієї роботи полягає в тому, щоб запропонувати *метод роботи* з проектування програмного забезпечення для керування кіберфізичними системами, і тим самим забезпечити вирішення проблеми зростаючої складності програмного забезпечення для керування. Цей метод роботи повинен бути універсально придатним для всіх видів кіберфізичних систем, таких як медичні системи, що пов'язані з хірургічними втручаннями, промислові системи, що використовуються для складальних ліній, сервісні роботи, що взаємодіють з людьми, або невеликі дослідницькі роботизовані системи. Метод роботи повинен охоплювати всю траєкторію проектування, від початкового проектування архітектури системи до розгортання програмного забезпечення для керування.

- Другою метою цієї роботи є надання плану для більш підходящого універсального програмного компонента, як того вимагає запропонований спосіб роботи. У попередніх роботах було зроблено спробу розробити такий компонент, але він не був достатньо гнучким та придатним для повторного використання, щоб зробити його придатним для різних типів кіберфізичних систем.
- Третя мета полягає в тому, щоб описати та надати структуру виконання та інструментарій, що відповідають методу роботи. Метод роботи наполегливо рекомендує використання моделей для проектування програмного забезпечення керування, оскільки це допомагає керувати складністю сучасних кіберфізичних систем. Ці моделі потребують структури та інструментарію, щоб стати повністю придатними для використання, що підвищує ефективність та зручність використання методу роботи [7].

Разом ці цілі повинні призвести до скорочення часу розробки (складного) програмного забезпечення для кіберфізичного керування. Деякі додаткові вимоги до цих цілей обговорюються в наступних розділах.

Дослідження простору дизайну

Протягом траєкторії розвитку кіберфізичної системи часто випробовується кілька варіантів проєктів, щоб знайти оптимальне рішення. Інші компоненти можуть бути тимчасово замінені прототипами, наприклад, з метою тестування або через відсутність готового компонента. Часткове використання прототипів для тестування називається апаратно-ін-тестуванням.

Спосіб роботи повинен підтримувати таку ітеративну траєкторію проектування. Наприклад, має бути можливість спробувати кілька реалізацій програмного контролера, щоб вони постійно поєднували різні механічні, прототиповані компоненти, або випробувати різні алгоритми керування, щоб побачити, який з них підходить. Тому змодельовані компоненти повинні бути взаємозамінними з подібними моделями компонентів. Розділення компонентів на інтерфейс та фактичну реалізацію допомагає у виконанні цієї вимоги, оскільки розробник може вибрати різну реалізацію для кожного інтерфейсу (компонента), що використовується в архітектурі системи.

Масштабованість

Спосіб роботи має бути масштабованим для траєкторій проектування всіх видів кіберфізичних систем: він повинен підтримувати як малі вбудовані кіберфізичні системи, так і великі кіберфізичні системи. Підтримка малих вбудованих систем полягає в тому, щоб враховувати використання ресурсів, щоб звести його до мінімуму [8]. Розробка програмного забезпечення для більших систем не вимагає суворого управління ресурсами, тому можна забезпечити додаткові функції та гнучкість.

Отже, спосіб роботи має бути гнучким, щоб підтримувати повний спектр траєкторій проектування, від проектування вбудованих систем до проектування великих систем. Це враховується у способі роботи шляхом надання додаткових кроків або шляхом здійснення необов'язкових кроків. Структура виконання повинна бути придатною для вбудованих систем з невеликим обсягом ресурсів, але також повинна забезпечувати додаткові функції та гнучкість, коли це необхідно. Ця вимога виконується шляхом створення модульної структури, тому функції можна вмикати або вимикати за бажанням, залежно від вимог системи, що розробляється.

З першого разу все гаразд

Запропонований метод роботи має бути надійним та призводити до розробки дизайну *з першого разу* . Рішення, яке вдається реалізувати з першого разу, було б ідеальним, але його майже неможливо досягти. Однак, підвищення продуктивності на 30% є можливим. Прагнення до рішення, яке вдається реалізувати з першого разу, є важливим, оскільки легше вносити зміни до корпоративного дизайну на початковому етапі розробки, ніж на пізнішому етапі, коли продукт вже продано. З точки зору компанії, час виведення на ринок має бути якомога коротшим, тому надійний метод роботи допомагає скоротити цей час.

Збільшення ймовірності правильного проєкту з першого разу досягається за допомогою методів модельно-орієнтованого проєктування (MDD). Моделі, що виникають в результаті використання цих методів, використовуються для всіляких завдань. Наприклад, їх можна формально перевірити або використовувати

для моделювання, щоб мінімізувати недоліки проєкту та виявити проблеми на ранній стадії [9]. Генерація коду з цих моделей, замість ручного надання коду, ще більше зменшує непередбачені проблеми. Все це підвищує ймовірність правильного проєкту з першого разу, зменшує необхідні зусилля налагодження та призводить до розробки програмного забезпечення, яке працює належним чином для кіберфізичної системи, для якої воно було розроблено.

Підхід

У цьому підрозділі наведено підхід до досягнення цілей. Він базується на структурному проектуванні програмного забезпечення керування для кіберфізичних систем. Це допомагає керувати зростаючою складністю цього програмного забезпечення керування та вирішує проблему.

Використання структурного підходу допомагає запобігти непотрібним помилкам або виявити їх на ранній стадії розробки, завдяки чому отримані кіберфізичні системи є максимально безпечними. Проект пропонує підхід, що базується *на формалізмах, техніках, методах та інструментах*. Він забезпечує розділення завдань під час проектування, оскільки кожна частина використовується окремо від трьох інших. Це розділення використовується протягом усієї роботи, щоб забезпечити загальні рішення, які не обмежуються певною мовою (формалізмом), технікою/методом проектування чи інструментом.

Методи проектування на основі моделей використовуються для моделювання загального огляду системи на ранніх стадіях та для забезпечення результуючої реалізації на пізнішому етапі. Для забезпечення семантики моделей використовуються два типи метамоделей: архітектурна метамодель для моделювання архітектури програмного забезпечення та метамодель комунікаційних послідовних процесів (CSP) для моделювання деталей реалізації. Розроблені моделі можна формально перевірити, змоделювати та використовувати для перетворень моделей. Безперервне повторне використання моделей є частиною структурного проектування та запобігає проблемам, пов'язаним з повторною реалізацією системи для іншого формалізму.

Описані методи модельно-орієнтованого проектування потребують

підтримки у вигляді фреймворку та інструментарію виконання. Самі моделі не можуть бути виконані безпосередньо, тому їх потрібно конвертувати в інший формалізм, тобто обчислювальну мову. Фреймворк виконання , для зручності, забезпечує статичну частину конвертованого формалізму, тому його не потрібно включати знову і знову зменшує складність процесу проектування, необхідна інструментальна підтримка . Редактори моделей надають підтримку в побудові моделей, що може бути виконано графічно або текстово. Інструменти трансформації перетворюють моделі з одного формалізму в інший, в результаті чого, наприклад, створюються моделі, оптимізовані для певної мети або на виконуваній комп'ютерній мові [10]. Інші інструментальні засоби підтримки надають засоби для перевірки, моделювання або керування моделями. Загалом, інструменти, що підтримують певний тип моделей, працюють разом, утворюючи набір інструментів, здатний забезпечити підтримку всієї траєкторії проектування.

Ця робота зосереджена на всіх цих аспектах, об'єднаних у запропонованому методі роботи, що описує необхідну траєкторію проектування програмного забезпечення для керування. Вимоги, згадані в попередніх розділах, включені в метод роботи, що робить його придатним для проектування програмного забезпечення для керування для всіх типів кіберфізичних систем.

1.2 Забезпечення роботи в режимі реального часу в кіберфізичних системах

Кожна кіберфізична система потребує своїх програмних компонентів, наприклад, тих, що містять алгоритми керування, які працюють на певній частоті для стабільної роботи. (Вбудоване) програмне забезпечення керування повинно відповідати вимогам щодо своєчасності кожного компонента і, таким чином, повинно мати результати роботи компонента готовими з необхідним інтервалом. Поділяється цю вимогу щодо своєчасності на дві підвимоги:

- *Вимога критичності*

Ця вимога визначає критичність проблеми у разі пропуску терміну. Для

виконання цієї вимоги система повинна надавати *гарантії в режимі реального часу* для своїх компонентів.

- *Вимоги до часу*

Ця вимога визначає необхідні часові періоди роботи компонентів, що визначає швидкість та швидкість реагування компонента.

Основна увага в цій роботи зосереджена на вимозі критичності та результуючих гарантіях реального часу, коли розглядається своєчасність компонента [11]. Вимога часу майже не впливає на проектування та виконання програмного компонента, вона має більше значення при проектуванні платформи програмних обчислень, яка використовується для виконання програмного забезпечення кіберфізичної системи.

Існує два різних типи гарантій реального часу: жорсткий реальний час та м'який реальний час. Визначається їхню різницю наступним чином:

«Якщо результат має корисність навіть після закінчення терміну, термін класифікується як *м'який* (...) Якщо ж пропуск терміну може призвести до катастрофи, термін називається *жорстким».*

Компоненти без однієї з цих двох гарантій роботи в режимі реального часу є некритично *важливими* компонентами, тобто вони не є частиною програмного забезпечення керування, але забезпечують інші, додаткові функції.

На рисунку 1.3 показано багаторівневу структуру вбудованого програмного забезпечення керування, що зображеного на рисунку 1.2. Кожен рівень, такий як інтерфейс користувача або керування послідовністю, використовується для певної мети та може містити один або декілька програмних компонентів. Кожен рівень програмного забезпечення вимагає принаймні одного типу гарантій реального часу, від повної відсутності реального часу до жорсткого реального часу. Однак більшість рівнів не мають суворої класифікації гарантій реального часу, але надають вибір своїм компонентам, що показано на рисунку вигнутими межами гарантій реального часу. Навіть у межах одного компонента можуть знадобитися дві різні класифікації.

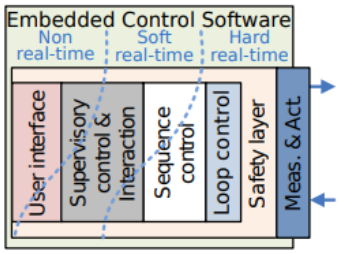

Рисунок 1.3 - Вбудована область програмного забезпечення керування кіберфізичною системою.

Рівень *керування циклом* – це програмна частина, яка безпосередньо відповідає за керування фізичною системою, і тому вона потребує жорстких гарантій реального часу. Гарантії жорсткого реального часу забезпечують суворі властивості синхронізації, гарантуючи, що задані терміни завжди дотримуються, за умови, що система має достатньо ресурсів для початку. Ці гарантії жорсткого реального часу необхідні для того, щоб розрахунки алгоритму керування були завершені вчасно, що загалом необхідно для підтримки стабільності періодичних циклів керування [12]. Тому, якщо це з якоїсь причини дає збій, система вважається небезпечною, і можуть статися катастрофічні аварії з фізичною системою або її оточенням через неправильно рухомі частини.

Керування послідовністю забезпечують підтримку для довготривалих завдань, таких як планування шляхів або картографування середовища. Залежно від вимог до програмного забезпечення, ці завдання потребують жорстких або м'яких гарантій виконання в реальному часі. Компоненти з м'якими гарантіями виконання в реальному часі намагаються дотримуватися своїх термінів, не надаючи жодних жорстких гарантій. Припускаючи, що проектування правильне, нічого поганого не повинно статися, якщо такий термін іноді не буде дотримано.

Рівень *контролю та взаємодії* містить компоненти, які потребують ще нижчих гарантій роботи в режимі реального часу порівняно з компонентами рівня *контролю послідовності*, такі як планування загальної діяльності або зв'язок з іншими (кіберфізичними) системами. Їхні алгоритми зазвичай працюють протягом тривалішого періоду часу через характер їхньої діяльності, і лише час від

часу їм потрібно змінювати або впливати на контролери послідовності для оновлення їхніх завдань. Ці компоненти забезпечуються або м'якими, або не гарантіями роботи в режимі реального часу. Залишкові ресурси системи використовуються для цих завдань без будь-яких гарантій їхньої доступності [13].

Рівень *інтерфейсу користувача* та певною мірою рівень *контролю та взаємодії* (частково) не працюють у режимі реального часу, оскільки вони раніше забезпечували результати, що безпосередньо впливають на критично важливі для безпеки частини системи. Як згадувалося раніше, ці межі гарантії реального часу не є суворими, наприклад, бувають ситуації, коли компонент потребує інших гарантій реального часу, ніж ті, що використовуються регулярно.

Жорсткі гарантії реального часу забезпечують найкращі гарантії дотримання заданих термінів, тому на перший погляд, з точки зору проектування, найпростішим рішенням було б забезпечити жорсткі гарантії реального часу для всього програмного забезпечення. На жаль, це неможливо для більшості систем, оскільки це занадто сильно навантажує використання ресурсів і вимагає значних зусиль розробника.

Залежно від класу кіберфізичної системи, вибір типу роботи в режимі реального часу різниться. Наприклад, інтерфейс користувача, який відображає інформацію журналу, може бути неважливим для гуманоїдного робота і може бути розміщений на типі, що не працює в режимі реального часу. Інтерфейс користувача промислового робота, ймовірно, важливіший і тому повинен мати м'які гарантії роботи в режимі реального часу, особливо якщо оператору потрібно, щоб він реагував на певні ситуації. З іншого боку, надання інтерфейсу користувача промислового робота жорстких гарантій роботи в режимі реального часу фактично є марною тратою ресурсів, оскільки оператор-людина також не реагує з жорсткими гарантіями роботи в режимі реального часу.

Через відмінності в гарантіях термінів виконання компонентів реального часу, зв'язок між компонентами різних типів реального часу не є простим. Властивості обох рівнів необхідно підтримувати під час передачі даних через межі рівнів. Наприклад, канал зв'язку між компонентом м'якого реального часу та

компонентом жорсткого реального часу може порушити гарантії компонента жорсткого реального часу [14]. Це трапляється, коли компонент жорсткого реального часу повинен чекати на доступність даних, оскільки компонент м'якого реального часу може не відповідати своїм гарантіям, що не є проблемою для *цього компонента, але* пропуск терміну виконання компонентом жорсткого реального часу *є* проблемою.

Простим рішенням є використання буфера, розташованого на межі між двома типами даних реального часу. Компонент м'якого реального часу оновлює буфер, коли його дані доступні або обчислюються. Компонент жорсткого реального часу зчитує дані з буфера, який завжди доступний і може бути прочитаний детермінованим чином. Те саме стосується випадків, коли компонент жорсткого реального часу повинен надсилати дані до компонента м'якого реального часу. Під час використання такого рішення алгоритми (керування) повинні приймати значення, які можуть ще не бути оновлені, інакше ця схема все одно призведе до системних збоїв.

Безпека

Навіть попри те, що програмне забезпечення для керування розроблено максимально ретельно та ретельно протестовано, заходи *безпеки* все ще необхідні для обробки неочікуваних та небажаних ситуацій. Ці ситуації можуть бути спричинені навколишнім середовищем або поспішними виправленнями, які були необхідні. Фактично, аспекти безпеки кіберфізичної системи дуже важливі, тому їх необхідно впроваджувати в різних областях, щоб у разі збою однієї області інша все ще могла забезпечити заходи безпеки.

Неможливо охопити всі проблеми безпеки лише шляхом ретельного тестування, наприклад, причини, пов'язані з навколишнім середовищем, важко передбачити та повністю охопити. Система повинна реагувати на такі ситуації, щоб запобігти аваріям із кіберфізичною системою та/або її середовищем. Архітектура програмного забезпечення для керування роботами відіграє велику роль у правильному обробці таких ситуацій [15].

Якщо припустити, що програмне забезпечення керування складається з

мережі компонентів, проблему можна вирішити на локальному рівні, глобальному рівні або на обох. Це залежить від походження, серйозності та інших властивостей кожної проблеми. *Локальне вирішення проблеми,* у компоненті, який виявив проблему, може запобігти повному вимкненню системи. Прикладами того, що можна вирішити локально, є:

- зламаний фізичний компонент, який не потрібен для базового функціонування системи.
- Датчик більше не повертає правильні значення вимірювань, але ці вимірювання можуть бути компенсовані іншими датчиками
- небажана область, в яку збирається увійти роботизована рука

Глобальна обробка помилок – це інша можливість, яка потрібна в більш серйозних ситуаціях, які вже неможливо вирішити локально. Наприклад, якщо вийшло з ладу кілька важливих фізичних компонентів, систему необхідно безпечно вимкнути, не збільшуючи пошкодження.

Локальна обробка помилок є кращою, оскільки локальна обробка безпеки якомога менше впливає на решту системи, дозволяючи решті системи продовжувати працювати. Тим не менш, більшість кіберфізичних систем, ймовірно, матимуть гібридне рішення для локальної та глобальної обробки помилок.

1.3 Модельно-орієнтована розробка програмного забезпечення кіберфізичних систем

Ця робота в основному зосереджена на методології розробки програмного забезпечення методом *модельно-керованої розробки* (MDD). Модельно-керована розробка також відома як модельно-кероване проектування або модельно-керована інженерія (MDE). Різниця між цими трьома термінами є незначною для використання в цій роботі, якщо взагалі є якась різниця.

Описується MDE як метод

«(Зменшити) розрив між проблемою та реалізацією програмного забезпечення, використовуючи систематичні перетворення від абстракцій рівня

проблеми до програмних реалізацій»

Крім того, вони стверджують, що складність проектування програмного забезпечення для систем зменшується

«завдяки використанню моделей, що описують складні системи на кількох рівнях абстракції та з різних точок зору, а також завдяки автоматизованій підтримці перетворення та аналізу моделей»

По суті, MDE (а отже, і MDD) – це інженерна техніка, яка використовує побудову моделі та перетворення на основі моделі для досягнення бажаного кінцевого результату. У цій роботі цим кінцевим результатом є (вбудоване) програмне забезпечення керування для кіберфізичних систем.

Моделі є основою методології MDD, вони використовуються для виконання всіляких складних або трудомістких завдань, які в іншому випадку розробник повинен виконувати вручну. Типовими прикладами таких завдань є перетворення моделей або (ко-)симуляції.

Потреба в перетвореннях моделі зростає, наприклад, коли модель можна сформулювати формальною мовою [16]. Після перетворення на таку формальну мову питання якості та узгодженості моделі можна ретельно перевірити за допомогою інструменту формальної верифікації. Інструмент *уточнення відхилень відмов* (FDR2) – це такий інструмент, який здатний формально перевірити алгебру процесів CSP. Отже, вихідну модель необхідно перетворити на алгебру CSP, щоб використовувати можливості FDR2. Звичайно, принципи, що лежать в основі вихідної моделі, повинні бути сумісними з алгеброю CSP, щоб мати змогу перетворити модель. Більше інформації про CSP та похідні від нього моделі.

Під час моделювання кіберфізичної системи використовуються різні типи моделей. Кожен тип моделі має свої переваги та недоліки і підходить для конкретного завдання або області. Перетворення моделей використовуються для перетворення цих різних типів моделей у форму, сумісну з усіма типами моделей, з метою їх поєднання [17]. Наприклад, перетворення моделі в код використовується для отримання фактичного програмного забезпечення керування кіберфізичною системою.

Методи проектування на основі моделей зазвичай поєднуються з інструментальною підтримкою, що забезпечує засоби для побудови необхідних моделей або виконання завдань (трансформації) за допомогою цих моделей. Інтеграція цих інструментів у набір інструментів (також званий ланцюжком інструментів) та подальша автоматизація цих завдань ще більше спрощує процес розробки, запобігаючи непотрібним помилкам, пов'язаним з людиною, та скорочуючи час розробки програмного забезпечення керування. Це робить невеликі та швидкі ітераційні цикли доступнішими, оскільки запроваджена ручна праця з виконання циклу, така як тестування, перевірка, моделювання тощо, виконується за допомогою набору інструментів підтримки MDD.

TERRA є прикладом такого набору інструментів MDD і є частиною цього дослідження. Детальна інформація про TERRA наведена в розділі, особливо рисунок корисний у цьому контексті, оскільки він показує, як набір інструментів (MDD) працює навколо моделей. *20-sim* – ще один приклад інструменту MDD, який надає засоби для проектування, аналізу та моделювання моделей мехатронних систем. Ці моделі варіюються від заводу до законів керування або їх комбінації. 20-sim надає засоби для генерації коду C++ моделей для їх вбудовування в програмні додатки для керування.

Моделі, побудовані та використані за допомогою методів MDD, можуть бути представлені у візуальному/графічному або текстовому вигляді. Обидва способи мають свої переваги та недоліки. Основним недоліком візуального моделювання є те, що (складна) діаграма може інтерпретуватися кожною людиною по-різному, часто залежно від навичок цієї людини. Візуальні моделі також вимагають додаткової інформації для візуального аспекту, таким чином захаращуючи інформацію «чистої моделі».

З іншого боку, візуальні моделі мають обмежений набір символів, на відміну від текстової моделі, яка може складатися з будь-якої кількості символів (слів, утворених окремими літерами), що зручно для початківців-моделерів [18]. Під час використання візуальних моделей ретельне проектування візуального представлення та символів покращує розуміння моделей. Це обговорюється для

діаграм UML, оскільки автори стверджують, що візуальні представлення діаграм UML містять недоліки, але це обговорення стосується візуальних представлень моделей загалом. Будь-який спосіб моделювання та представлення результуючих моделей застосовний при використанні методів MDD і не має (великого) значення в рамках цієї роботи.

Методології MDD, як описано вище, неминучі для відповідності як технічним, так і бізнес-потребам сучасних проектів. Контроль якості та автоматична перевірка узгодженості є тут вирішальними для підтримки ефективного процесу проектування. Використання інструментів MDD робить розробку складних кіберфізичних систем менш складним та більш зручним у обслуговуванні завданням.

Кінцева мета MDD - створювати проєкти, які є правильними з першого разу, тобто мати модель(и), перевірені на кількох етапах процесу проєктування, реалізація (код) коректна відповідно до специфікації (моделі) та задовольняючи всі вимоги, поставлені в проєкті.

Мета-моделі

Методологія розробки програмного забезпечення MDD широко використовує моделі: усі супутні інструменти вимагають розуміння цих моделей. Тому моделі повинні відповідати визначенню, щоб їхній зміст розумівся правильно та так, як задумано. Таке визначення може існувати лише в голові розробника інструменту та безпосередньо реалізується в інструментах. Великим недоліком є те, що це визначення побудовано нечітко, тобто, коли потрібні додаткові дані моделі, розробник просто додає їх до визначення, а інструменти, яким потрібні ці додаткові дані моделі, оновлюються. Після кількох таких ітерацій визначення моделі в голові розробника стає більш розмитим і все більше відхиляється від реалізації в різних інструментах. Звичайно, коли присутні кілька розробників, визначення моделі стає ще більш розмитим.

Те саме стосується повторного використання моделей та їх визначень, отриманих за допомогою сторонніх інструментів. Їхня семантика моделей не є загальнодоступною або існує лише в головах розробників інструментів. Це обмежує розширюваність та прозорість моделей, що використовуються цими

інструментами [19]. Тому в цьому дослідженні не буде використана неявна інформація про моделі цих сторонніх інструментів для цілей моделювання, щоб запобігти (майбутнім) проблемам.

метамоделі чітко визначають структуру моделі. Метамодель – це, по суті, модель, яка описує семантику *цільової моделі,* звідси й назва метамодель. Це можна продовжити до будь-якого бажаного рівня, наприклад, метаметамодель описує семантику метамоделі.

Оскільки метамоделі є, по суті, моделями, їх також можна розробляти за допомогою методології MDD. Наприклад, EMF надає метамодель Ecore, яку можна використовувати для моделювання власних метамоделей. Ці метамоделі можна використовувати як основу для графічних редакторів та інших інструментів на основі моделей. Метамодель Ecore є крайнім прикладом метамоделювання: метамодель Ecore описується за допомогою самої себе як метамоделі.

За допомогою інструментів MDD метамодель можна перетворити на програмне забезпечення, яке може використовуватися інструментами MDD, яким потрібно розуміти моделі, описані метамоделлю. Зазвичай, але принаймні у випадку EMF, згенероване програмне забезпечення містить засоби для зберігання та отримання моделі. Через суворість метамоделі, а отже, і програмного забезпечення, обробляються лише дані, визначені метамоделлю, що призводить до чистої моделі без невизначених даних.

Порівняно з реалізацією типу «визначення моделі в голові розробника», інструменти, що використовують згенеровану метамодель програмного забезпечення, безпосередньо використовують похідну версію самої метамоделі. Будь-які зміни в метамоделі відображаються в результуючому програмному забезпеченні, а отже, і в інструментах. Якщо таке визначення змінюється, всі інструменти автоматично використовують найновішу версію. Навіть якщо інструмент не потребує оновлених визначень метамоделі, він все ще здатний обробляти дані моделі без змін. Отже, після оновлення метамоделі всі інструменти MDD здатні використовувати нові визначення моделі без особливих зусиль.

Підключення компонентного порту

Обговорення послідовних процесів

У підрозділі представлено *Послідовні процеси комунікації* (CSP) та показано, що описувати комунікаційні потоки в рамках моделі доцільно. Опис цих комунікаційних потоків – це саме те, що потрібно при моделюванні архітектури кіберфізичної системи.

Крім того, CSP надає можливість синхронізувати процеси на основі їхнього потоку зв'язку. Наприклад, коли *процес А* вимагає певного значення від *процесу В* , можна використовувати канал *зв'язку "рандеву"* , щоб переконатися, що процес А блокується, доки процес В не зможе надати необхідне значення. Іншим рішенням було б розмістити обидва процеси в послідовному порядку, що гарантує, що процес В завершиться раніше за процес А, а значення буде доступним під час виконання процесу А. *Буферизовані* канали зв'язку є протилежністю каналам зв'язку "рандеву": вони не забезпечують синхронізацію, оскільки дані буферизуються в каналі. Як процес читання, так і процес запису можуть взаємодіяти з каналом, не будучи блокованими. Цей тип каналу зазвичай використовується для зв'язку між компонентами, які мають різні гарантії реального часу.

Крім того, CSP бездоганно дотримується парадигми CPC. Процеси CSP можна розглядати як компоненти, а канали CSP – як зв'язок між цими процесами. Порти CPC можна розглядати як частину процесу CSP, який визначає, які канали потрібно підключити до процесу. Наприклад, процес *зчитування* або *запису* зчитує або записує дані з каналу відповідно. Тому обом потрібен порт для підключення каналу. Метамоделі TERRA, містять елементи порту саме з цієї причини.

Ще однією перевагою використання CSP як методології моделювання є те, що CSP підходить для формальної перевірки коректності моделі. *Формальна верифікація* моделі надає уявлення про проблеми моделювання, такі як глухі блокування або активні блокування. FDR2 – це інструмент, який здатний формально перевіряти (машиночитані) моделі CSP на наявність таких проблем.

Взаємоблокування – це недоліки проектування, коли два або більше компонентів нескінченно чекають один на одного, щоб надати інформацію для продовження виконання, і ця частина програми зупиняється. Активні блокування –

це недоліки проектування, коли група процесів ніколи не припиняє виконання, що запобігає виконанню інших процесів, що називається голодуванням.

На рисунку 1.5 показано простий приклад тупикової ситуації. На ньому показано дві послідовні групи, що містять процес запису (коло зі знаком оклику) та процес читання (коло зі знаком питання). Послідовна група визначає порядок виконання процесів, які вона містить. Стрілка на рисунку показує цей порядок, в обох послідовних групах спочатку виконується процес запису, а потім процес читання [22]. Канали CSP між парами запису/читання є каналами зустрічі. Як згадувалося раніше, процеси обох кінців таких каналів повинні бути активними, щоб мати змогу обмінюватися даними. Через блокування записів в обох послідовних групах, зчитувачі ніколи не стають активними, що призводить до тупикової ситуації. Причина тупикової ситуації в цьому прикладі дуже очевидна і її можна легко знайти. Простим рішенням для запобігання цій тупиковій ситуації є заміна зчитувача та запису однієї з послідовних груп або зміна послідовних груп на паралельні групи [23]. Для складніших моделей використання формальних інструментів перевірки забезпечує засоби для пошуку подібних проблемних ситуацій, які інакше не були б помічені.

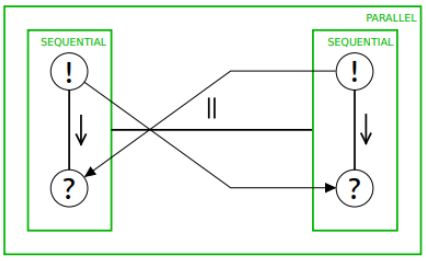

Рисунок 1.5 - Приклад очевидної ситуації блокування

(Математичну) нотацію CSP неможливо записати з використанням кодування ASCII, тому неможливо використовувати цю нотацію як вхідні дані для інструментів, пов'язаних з CSP. Тому розроблено машинозчитуваний CSP. Він використовує набір символів ASCII для опису моделей CSP. Наприклад, машинозчитуване представлення CSP попереднього рисунка виглядає наступним

чином:

Лістинг 1.1 - Машинозчитуване представлення CSP

```
channel c1, c2
SEQ1 = c1!var_writer1 ; c2?var_reader1
SEQ2 = c2!var_writer2 ; c1?var_reader2
PAR = SEQ1 [| {| c1, c2 |} |] SEQ2
```

Трансформації моделей

Метамоделі та варіанти їх використання обговорюються в попередніх розділах. Сучасні набори інструментів MDD надають графічні редактори для ручного конструювання моделей на основі цих метамоделей. Іншим методом побудови моделі є *перетворення моделі в модель* (M2M): вихідна модель використовується для побудови результуючої моделі. Отримана модель може відповідати або тій самій метамоделі, що й вихідна модель, або іншій метамоделі. Прикладом першого є *оптимізація моделі,* вихідна модель оптимізується з акцентом на певний аспект, а прикладом останнього є перетворення з однієї області в іншу. Приклад реалізації та додаткові деталі перетворення моделі в модель.

Перетворення тексту в модель (T2M) – це ще один спосіб побудови моделей, його можна використовувати для (пере)творення моделі з текстової інформації, наприклад, вихідного коду, щоб зробити її структуру зрозумілішою. Наприклад, код C++ можна перетворити на модель CSP, щоб отримати (краще) розуміння паралельної архітектури коду. Оскільки файли коду не містять усієї необхідної інформації, яку вимагають (графічні) моделі, як-от координати та розміри об'єкта, її потрібно додати або алгоритму перетворення, або користувачеві. Така форма побудови моделі зустрічається нечасто, ймовірно, через проблеми складності, відсутність даних моделі та відсутність функціонального використання.

Перетворення моделі також використовуються для перетворення моделі на кінцевий, необхідний об'єкт. Модель використовується як простий засіб для побудови або збору необхідної інформації для цього кінцевого об'єкта.

Наприклад, модель CSP може бути перетворена на машинозчитуваний файл CSP, щоб його можна було використовувати формальними інструментами верифікації, такими як FDR2 [24]. Це перетворення моделі називається *перетворенням моделі в текст* (M2T), оскільки результуючий файл є звичайним текстовим файлом. Якщо результат перетворення має форму комп'ютерної мови або вихідного коду, перетворення також називається *перетворенням моделі в код* (M2C) або *генерацією коду.* Різниця між методами перетворення M2T та M2C мінімальна, якщо взагалі є.

Спільне моделювання

Формальна перевірка моделі показує лише проблеми архітектурного проектування. Потрібні додаткові перевірки, щоб побачити, чи модель поводиться належним чином [25]. Виконання програмного забезпечення безпосередньо на реальній кіберфізичній системі може бути небезпечним. Особливо, коли можуть виникнути небезпечні екологічні ситуації або система пошкодиться через неправильне та непротестоване програмне забезпечення. *Програмне моделювання* надає засоби для безпечної перевірки того, чи містять архітектурна модель та її реалізовані компоненти помилки.

Моделі, отримані в результаті використання методів MDD, також можуть бути використані для симуляцій. Ці моделі базуються на метамоделях, тому визначення всіх елементів моделі відомі. Симулятор використовує ці відомі визначення, щоб визначити, як поводиться програмне забезпечення під час виконання на реальній обчислювальній платформі. Наприклад, метамодель CSP неявно містить визначення порядку виконання процесів. Ці визначення можуть бути використані симулятором для визначення потоку виконання моделі CSP. Фактична точність симуляції залежить від симулятора і може варіюватися від простого визначення порядку виконання процесів до повної симуляції, ніби програмне забезпечення виконувалося б на певній апаратній платформі.

Описані симуляції практично непридатні для моделей, що містять компоненти, реалізація яких забезпечується зовнішніми інструментами, оскільки потрібні знання про формат та поведінку цих реалізацій, тобто визначення цих

метамоделей невідомі симулятору. Цю проблему можна подолати, коли зовнішній інструмент має можливості симуляції, доступ до яких можна отримати ззовні інструменту. У цих випадках необхідно використовувати можливості симуляції інструменту, тому явне знання зовнішньої метамоделі не потрібне. Цей метод поєднання симуляторів кількох інструментів називається *ко-симуляцією.*

На рисунку 1.6 показано приклад косимуляції. Один інструмент повинен взяти на себе ініціативу під час косимуляції, цей інструмент у прикладі називається Головним інструментом, забезпечуючи синхронізацію всіх інструментів моделювання. Синхронізація між інструментами моделювання полягає, наприклад, у видачі команд для виконання одного кроку моделювання іншим симуляторам, що забезпечує правильний порядок виконання. Іншим завданням синхронізації є надання необхідних даних інструменту моделювання, який збирається виконувати моделювання [26]. У прикладі процеси A1 та A2 потребують даних від P1, тому Головний інструмент повинен надати ці дані Інструменту A, щоб Інструмент A міг моделювати наступний крок. Результат цього моделювання потім надсилається до P2, який надсилає свій результат до іншого зовнішнього Інструменту B.

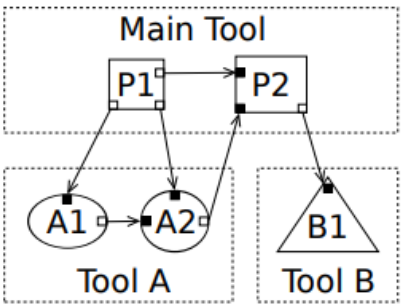

Рисунок 1.6 - Приклад косимуляції

Провідним інструментом зазвичай є інструмент, який виконує моделювання архітектурної моделі системи. Він містить інформацію про зовнішні змодельовані компоненти, необхідну для синхронізації. Ця інформація необхідна для виконання спільного моделювання, оскільки вона визначає порядок виконання на найвищому рівні системи.

Варіантом ко-симуляції є *симуляція з використанням апаратного забезпечення в циклі*. Як вже вказує термін, цей тип симуляції (частково) використовує фактичне обладнання для виконання симуляцій, а не зовнішній інструмент симуляції. Цей тип симуляції особливо корисний у ситуаціях, коли частини апаратного забезпечення доступні, а інші – ні. Перевагою над звичайними ко-симуляціями є те, що доступне обладнання можна використовувати для отримання більш реалістичних результатів вищої якості. Симуляції з використанням апаратного забезпечення економлять час розробки, оскільки розробникам не потрібно чекати, поки буде побудована та доступна повна кіберфізична система, але все одно вони можуть підвищити точність симуляцій і, таким чином, якість програмного забезпечення.

Ще однією перевагою моделювання апаратного забезпечення в циклі є те, що систему можна додатково перевірити в безпечному середовищі [27]. Наприклад, обмеження часу програмного забезпечення можна виміряти та перевірити, використовуючи фактичну цільову платформу, але все ще використовуючи моделювання заводу. У ситуаціях, коли ці обмеження ще не дотримані, що призводить до проблемних ситуацій, фактичний план t не пошкоджується.

Розгортання

Після того, як формальні перевірки та (ко-)симуляції показали, що модель розроблена та працює належним чином, її необхідно розгорнути на цільовій платформі. Очевидно, що модель не може бути виконана на цільовій платформі безпосередньо , оскільки її потрібно перетворити на виконуваний код, який розуміє цільова платформа.

Спочатку модель перетворюється на *вихідний код* за допомогою перетворення моделі в текст , який потім потрібно скомпілювати *у* виконуваний додаток. Компіляція вихідного коду для іншої цілі (порівняно з середовищем розробки) називається *крос-компіляцією.* Кіберфізичні системи здебільшого мають вбудовані комп'ютери, які мають мало ресурсів і зазвичай не мають (багато) підключених периферійних пристроїв. Через брак ресурсів ці вбудовані комп'ютери зазвичай не мають встановлених інструментів розробки. Тому для розробки

вбудованого програмного забезпечення керування потрібен комп'ютер розробника, а крос-компіляція необхідна для створення фактичного виконуваного додатку для вбудованого комп'ютера.

Далі, скомпільований виконуваний файл потрібно перенести з комп'ютера розробника на цільове обладнання. Якщо це не потрібно надто часто, це можна зробити вручну, але здебільшого це розгортання потрібно виконувати кілька разів, а інструменти розгортання надають послуги для спрощення цього повторюваного завдання.

Пакети інструментів розгортання обробляють крос-компіляцію, передачу виконуваного файлу на цільове обладнання та фактичне виконання на цільовому об'єкті. Інструмент розгортання також може надавати такі послуги, як збір даних журналів та моделювання апаратного забезпечення в циклі, що ще більше спрощує завдання розробника.

Очевидно, що набір інструментів розгортання потребує додаткової інформації для своїх завдань [28]. Наприклад, потрібна цільова інформація про апаратно-специфічні компоненти, тому її можна використовувати під час створення виконуваного файлу для конкретної цільової платформи. Такі деталі, що стосуються апаратного забезпечення, можуть надати інформацію про доступ до датчиків та виконавчих механізмів кіберфізичної системи з програмного забезпечення. Фактично необхідні деталі залежать від інструменту розгортання та послуг, які він надає.

Програмні фреймворки

Інструменти перетворення моделі в код здатні конвертувати модель у вихідний код. Фактичне виконання моделей вимагає багато додаткового коду для підтримки всіх використовуваних елементів метамоделі. Наприклад, процес CSP вимагає *механізму виконання*, який планує процеси у правильному порядку виконання. Використовувані елементи каналу вимагають фактичних реалізацій для фактичної надсилання даних від процесів, що їх виробляють, до процесів, що їх споживають. Ці необхідні функції також можуть бути забезпечені перетвореннями, але оскільки це здебільшого статичний код, має більше сенсу

використовувати програмний *фреймворк*, який надає цей необхідний статичний код. Це спрощує перетворення та спрощує підтримку статичного коду.

Окрім надання високорівневих компонентів, таких як механізм виконання, фреймворк також використовується для забезпечення операційної системи та/або цільової незалежної програмної платформи за допомогою *рівня апаратної абстракції* (HAL). Зазвичай він приховує платформозалежні системні виклики за допомогою публічного *інтерфейсу програмування додатків* (API). API надає інтерфейс, який має одну або кілька реалізацій, прихованих від програми. Залежно від певних властивостей, у цьому випадку вибраної цільової платформи, використовується правильна реалізація без

активної взаємодії програми [29]. Це призводить до того, що платформозалежна програма не знає про базову ОС або апаратне забезпечення. В ідеалі така програма повинна виконуватися на різних платформах без необхідності застосування платформозалежних модифікацій.

1.4 Висновок по розділу

У першому розділі було розглянуто теоретичні основи кіберфізичних систем та особливості їх функціонування. Проаналізовано принципи взаємодії програмних і фізичних компонентів, що забезпечують обробку даних та керування процесами в режимі реального часу. Досліджено механізми забезпечення роботи кіберфізичних систем із дотриманням вимог до швидкодії, надійності та синхронізації. Також розглянуто модельно-орієнтований підхід до розробки програмного забезпечення, який дозволяє підвищити якість проєктування та спростити процес створення складних систем. Отримані результати формують теоретичну основу для подальшого дослідження методів і засобів розробки програмного забезпечення кіберфізичних систем.

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КІБЕРФІЗИЧНИХ СИСТЕМ

2.1 Підходи до проєктування програмного забезпечення вбудованих систем керування

Шаблони проектування допомагають керувати складністю проектування (вбудованого) програмного забезпечення керування для сучасних кіберфізичних систем. Спосіб *роботи* , запропонований у цьому розділі, використовує підхід до модельно-орієнтованої розробки в поєднанні з розділенням завдань для зменшення складності. Він охоплює повну траєкторію проектування, починаючи з початкового проектування архітектури системи до розгортання готового програмного забезпечення керування на цільовому об'єкті.

Кінцева мета цього методу роботи - забезпечити *правильний підхід з першого разу* до програмної реалізації кіберфізичної системи. Запобігання надмірному часу проектування, водночас збереження точки зору дизайнера та бажаних методів і інструментів, є більш реалістичною метою цього методу роботи. Це досягається шляхом визначення структурованого способу проектування програмного забезпечення, повторного використання раніше розроблених моделей та компонентів, що ще більше скорочує час проектування.

Досліджується кілька методів проектування, які здаються перспективними для підвищення якості та скорочення часу проектування програмного забезпечення для керування [30]. Наприклад, проект BRICS визначив *модель компонентів BRICS* (BCM) та відобразив її на 4 рівнях абстракцій моделі (метамоделей), визначених OMG. Проект DESTECS зосереджений на спільному моделюванні та управлінні моделями. Спосіб роботи повинен інтегрувати або принаймні толерувати ці досліджені методи проектування, щоб зробити спосіб роботи максимально універсальним та застосовним.

Спосіб роботи

На рисунку 2.1 показано необхідні кроки розробки програмного забезпе-

чення під час розробки вбудованого програмного забезпечення керування для кіберфізичної системи. Починається проектування архітектури *програмного забезпечення (Ia)* та *алгоритмів керування (Ib).* Обидва варіанти зазвичай проектуються за допомогою інструментів модельно-орієнтованої інженерії та призводять до створення моделей архітектури та алгоритмів. За допомогою методів верифікації та моделювання (*II*) ці моделі постійно вдосконалюються, доки не буде задоволено їх необхідну функціональність.

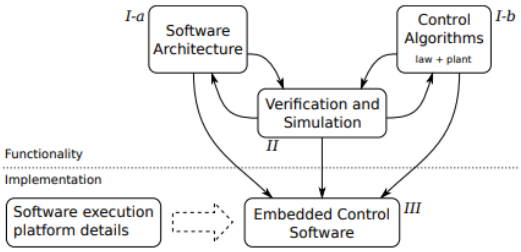

Рисунок 2.1 - Огляд необхідних кроків для розробки вбудованого програмного забезпечення для керування кіберфізичними системами

Вбудоване програмне забезпечення керування створюється шляхом поєднання архітектури програмного забезпечення та моделей алгоритмів законів керування *(III).* Додаткові деталі реалізації, такі як деталі обчислювальної платформи, датчиків та виконавчих механізмів, додаються для того, щоб програмне забезпечення могло виконуватися на кіберфізичній системі.

Більш детальні кроки проектування програмного забезпечення для керування показано на рисунку 2.2. Рисунок містить 5 відправних точок (з 5 доступних гілок проектування), позначених від (a) до (e). Кожна відправна точка охоплює певну область проектування. Ефективне проектування програмного забезпечення для керування повною мірою використовується, коли певні аспекти програмного забезпечення розробляються одночасно. Така одночасна розробка гарантує, що проблеми реалізуються в найбільш підходящій області.

Електрична *область* (a) та *механічна область* (e) виходять за рамки розробки програмного забезпечення для керування та далі не обговорюються. Гілки

проектування програмного забезпечення та контролерів ((b), (c) та (d)) відповідають двом відправним точкам *(Ia* та *Ib)* , показаним на рисунку 2.1.

Залежно від цілей та інтересів розробника, гілка (b) або (d) є найімовірнішими відправними точками, оскільки гілка (c) вимагає інформації з гілки (d), яка ще не доступна. Обидві гілки (b) та (d) можна одночасно використовувати для початку розробки (керуючого) програмного забезпечення, якщо є достатньо велика команда розробників.

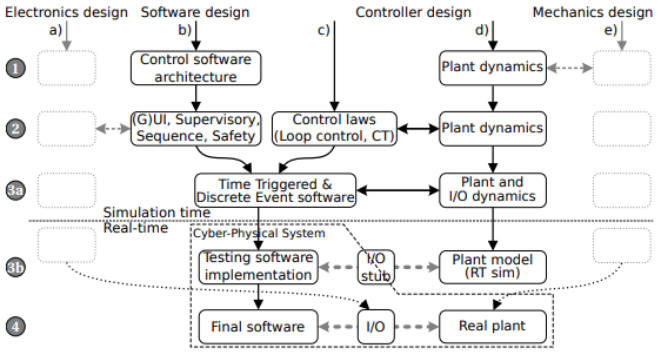

Рисунок 2.2 - Етапи роботи з розробки програмного забезпечення для керування кіберфізичними системами

На певних етапах відбувається взаємодія між різними гілками розробки, що зображено на рисунку горизонтальними двонаправленими стрілками. У цих точках проекти в різних областях «синхронізуються»; характеристики обох моделей порівнюються. Оскільки кожна область має свої унікальні властивості, проблеми проектування можуть бути вирішені або спрощення можуть бути отримані шляхом вибору правильної області під час реалізації певного аспекту програмного забезпечення. Наприклад, небажану динамічну поведінку можна вирішити як в області контролера, так і в області механіки, точка взаємодії у верхньому правому куті рисунка забезпечує засоби для переміщення проблеми до області, яка найбільше підходить для її вирішення [31].

Гілка (b) є найбільш цікавою для розробки програмного забезпечення для кіберфізичних систем, тому саме ця гілка буде основною темою подальшого обговорення. Крім того, в передбачалося, що завод та його моделі вже доступні або

розробляються іншими членами команди.

Спосіб роботи складається з наступних кроків, позначених сірими кружечками на рисунку. Ці кроки є необов'язковими, якщо один з них не потрібен для проектування програмного забезпечення кіберфізичної системи, його можна пропустити, але для того, щоб проектувати програмне забезпечення правильно з першого разу, рекомендується використовувати всі кроки:

Програмна гілка (b) починається з розробки архітектурного огляду всіх компонентів керування циклом, необхідних для керування всіма частинами кіберфізичної системи.

До архітектурного огляду додано додаткові компоненти для обробки завдань високого рівня, таких як компоненти послідовного керування, диспетчерського керування або інтерфейсу користувача. Тепер конструкція виглядає подібно до частини вбудованого програмного забезпечення керування, показаної на рисунку 1.2. На цьому етапі гілка (c) вимагає проектування контролерів низького рівня (контур), що зазвичай виконується за допомогою динамічної моделі об'єкта гілки (d).

Реалізації компонентів керування циклом архітектурної моделі забезпечуються проектами законів керування гілки (c). Реалізації інших змодельованих компонентів також необхідно заповнити на даний момент. На цьому етапі змодельована функціональність програмного забезпечення керування кіберфізичною системою завершена. Динамічна модель об'єкта тепер включає динаміку вводу/виводу для імітації фактичних сигналів датчиків та виконавчих механізмів, доступних у кіберфізичній системі.

На цьому підкроці отримане програмне забезпечення (моделі) *(спільно) моделюється* у співпраці з динамічною моделлю установки, щоб перевірити, чи поводиться програмне забезпечення належним чином. Залежно від результатів моделювання, моделі можна точно налаштувати для досягнення кращої поведінки.

Коли точне налаштування програмного забезпечення завершено та показано бажану поведінку, обмеження в реальному часі включаються шляхом тестування програмного забезпечення керування на цільовій обчислювальній

платформі. Знову ж таки, результати моделювання та тестування можуть бути використані для подальшого налаштування компонентів програмного забезпечення.

Коли механічне, електронне та програмне проектування завершено та належним чином функціонує відповідно до симуляцій, програмне забезпечення можна розгорнути та протестувати на кіберфізичній системі.

Фаза моделювання (рис. 2.1, крок *I)* є найбільш цікавою для проектування програмного забезпечення для керування. Розділення специфікації компонентів та їх реалізації призводить до кращих повторно використовуваних компонентів. Спосіб роботи також практикує це розділення: специфікація компонентів керування обробляється на кроці (T), в результаті чого створюється архітектурна модель. Високорівневі компоненти додаються до цієї мережі на кроці (2). Тепер вона надає інформацію про вимоги до компонентів, що використовуються в системі, та показує мережу цих компонентів. Наступна частина кроку (2) та перша частина кроку O забезпечити реалізацію компонентів, які були визначені в архітектурній моделі.

Загальний огляд методу роботи, що включає всі сфери розробки та їх взаємодію, зображено на рисунку 2.2. У наступних розділах основна увага приділяється конкретним аспектам етапів гілки проектування програмного забезпечення на етапі моделювання, пояснюючи деталі, необхідні для проектування програмного забезпечення кіберфізичної системи відповідно до методу роботи.

2.2 Моделювання архітектури програмного забезпечення кіберфізичних систем

Архітектуру програмного забезпечення необхідно моделювати відповідно до методології CPC [32]. Окремі частини системи моделюються як компоненти з портами та з'єднаннями для їх зв'язку. Ці компоненти реалізують алгоритми контролера, необхідні для керування окремими частинами системи. До

моделі необхідно додати додаткові компоненти нагляду та послідовного керування , щоб забезпечити співпрацю компонентів контролера, щоб система могла виконувати складніші завдання.

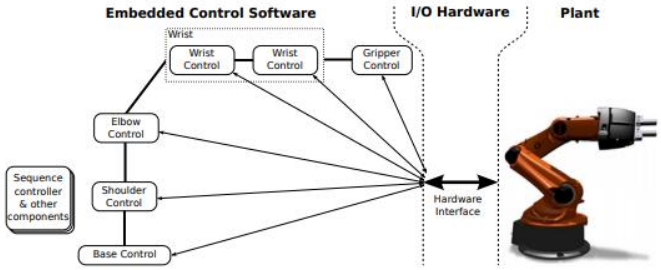

Рисунок 2.3 - Приклад огляду кіберфізичної системи

Приклад огляду архітектури кіберфізичної системи показано на рисунку 2.3, він відповідає огляду на рисунку 1.2. Праворуч зображено приклад установки, в цьому випадку маніпулятор youBot, він має 5 шарнірів, захоплювач і встановлений на основі. Посередині доступний апаратний інтерфейс, зображений у спрощеній формі за допомогою двонаправленої стрілки, оскільки його фактична реалізація виходить за рамки цієї роботи. Ліва частина рисунка показує всі окремі компоненти спільного керування з деякими додатковими компонентами завдань контролю та послідовного виконання.

Архітектурна модель програмного забезпечення керування кіберфізичною системою показана на рисунку 2.4. Усі визначені одиниці архітектурного огляду мають власний компонент, що містить реалізації законів керування. Ці контролери циклу керування потребують даних датчиків свого компонента з апаратного інтерфейсу та передають сигнали керування на апаратний інтерфейс, що призводить до типової структури циклу керування.

Компонент контролера послідовності використовується для керування контролерами циклу [33]. З'єднання між контролером послідовності та контролерами циклу використовуються для керування та синхронізації контролерів циклу з метою виконання фактично необхідних завдань за допомогою контролера ARM. Реалізація компонента апаратного інтерфейсу використовує доступні

фреймворки або драйвери, які здатні взаємодіяти з конкретними фізичними апаратними блоками.

Як згадувалося, кожен із суглобів має свій власний компонент в архітектурній моделі. Якщо припустити, що суглоби в цьому прикладі кіберфізичної системи керуються подібним чином, реалізації (деяких) компонентів суглобів можна використовувати повторно. Плечовий, ліктьовий та зап'ястний нахилювальні суглоби є кутовими обертовими суглобами з обмеженим діапазоном. Тому їхній компонент керування циклом повинен використовувати ті самі реалізації, забезпечені їхніми початковими параметрами для отримання необхідної динамічної поведінки для тієї частини роботизованої руки, якою їм потрібно керувати. Те саме стосується базового та зап'ястного поворотних суглобів, ці суглоби також можуть мати спільну реалізацію контролера.

Повторне використання можна зробити ще одним кроком, оскільки всі компоненти потребують базового набору функціональних можливостей. Базовий набір функціональних можливостей потрібен для того, щоб компоненти могли працювати поруч один з одним, об'єднувати їх у більші, складніші компоненти та взаємодіяти один з одним та з навколишнім середовищем. Усі компоненти також потребують певної форми підтримки безпеки, щоб реагувати на (не)передбачені проблеми безпеки, та підтримки конфігурації, тому параметри конфігурації можуть бути надані контролеру для забезпечення специфічної поведінки більш загальної, повторно використовуваної реалізації контролера [34]. Такі функціональні можливості компонентів за замовчуванням повинні надаватися шаблонним компонентом, що запобігає винаходженню велосипеда для кожного компонента та запобігає повторним зусиллям розробки.

Тестування програмного забезпечення

Після завершення фаз моделювання архітектури, кроки а) результати потребують більш ретельного тестування, як зазначено в кроці 0. Під час проектування моделей деякі функціональні тести, ймовірно, вже будуть виконані, цей крок також включає поведінкові тести для перевірки реалізації компонентів. Як згадувалося раніше в обговоренні мети «вперше правильно», ретельне

тестування моделей запобігає проблемам під час виконання, вирішення яких потім вимагає більше зусиль або може призвести до пошкодження (фізичних) частин системи.

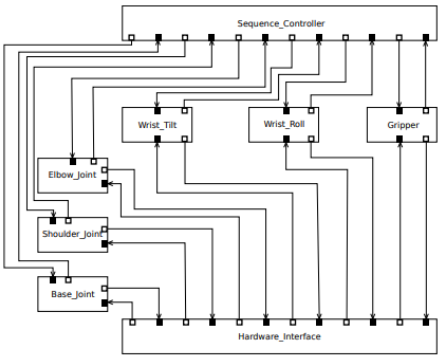

Рисунок 2.4 - Модель архітектури для прикладу кіберфізичної системи

Етап тестування поділяється на дві частини:

1. Крок O тестує моделі на рослинній рослині на більш архітектурному рівні, тобто перевіряє, чи програмне забезпечення поводиться очікувано.
2. Крок (3b) зосереджений на вимогах до часу, щоб перевірити, чи дотримано необхідних термінів .

Фактичні методології тестування залежать від типу моделі, яка потребує тестування, наприклад, необхідно перевірити, чи всі стани, що є частиною регулярного виконання, досяжні, а модель CSP необхідно перевірити за допомогою формальних методів верифікації, щоб перевірити наявність проблем з плануванням, таких як тупикові блокування, що призводять до зависання програми. Крім того, методології тестування можуть або перевіряти функціональну коректність, наприклад, формальні перевірки, або чи поводиться програма належним чином [35]. Останнє здебільшого виконується за допомогою *візуальної перевірки* розробником, оскільки важко забезпечити поведінкові тести, які перевіряють повну поведінку системи. Візуальна перевірка можлива за допомогою косимуляції та динамічних моделей об'єктів, що запобігає поломці фактичної системи, коли поведінкові тести не вдалися, а розробник не вимкнув систему вчасно.

Розгортання програмного забезпечення

Програмне забезпечення можна виконати на цільовій платформі після проходження тестування програмного забезпечення. Моделі необхідно перетворити на виконуваний додаток, який можна розгорнути на *обчислювальній платформі.* Перетворення та розгортання програмного забезпечення на цільовій платформі вимагає виконання кроків, показаних на рисунку 2.5, разом ці кроки утворюють крок (4) .

Отримані моделі кроку (3) цього способу роботи показано у верхній частині рисунка. Для проектування цих моделей можна використовувати один або кілька інструментів, і вони можуть взаємодіяти один з одним. Точні деталі залежать від інструментів, що використовувалися в процесі проектування, та вимог програмного забезпечення керування. Перетворення моделей на виконуваний код на цільовій платформі виконується за допомогою таких кроків:

1. Ці моделі спочатку потрібно конвертувати у вихідний код. Перед цим перетворенням може відбутися рефакторинг моделі з використанням перетворень моделі до моделі з міркувань оптимізації моделі . Генерація коду виконується самим інструментом моделювання або тісно інтегрованою частиною набору інструментів чи ланцюжка інструментів, частиною якого також є інструмент моделювання.

2. 1. Специфічна для цілі інформація додається до вихідного коду цільовим конектором. Ця інформація зазвичай містить деталі про цільову кіберфізичну систему, на якій розміщено програмне забезпечення керування, такі як можливості вводу/виводу та інша інформація про (периферійне) обладнання.. Як і раніше, це, ймовірно, залежить від інструменту моделювання та від того, чи є він частиною більшого набору інструментів.

3. Тепер код потрібно скомпілювати для створення фактичного виконуваного застосунку, який буде виконуватися на цільовому комп'ютері. Під час компіляції зазвичай додається один або декілька фреймворків для забезпечення механізмів виконання та рівнів апаратної абстракції.

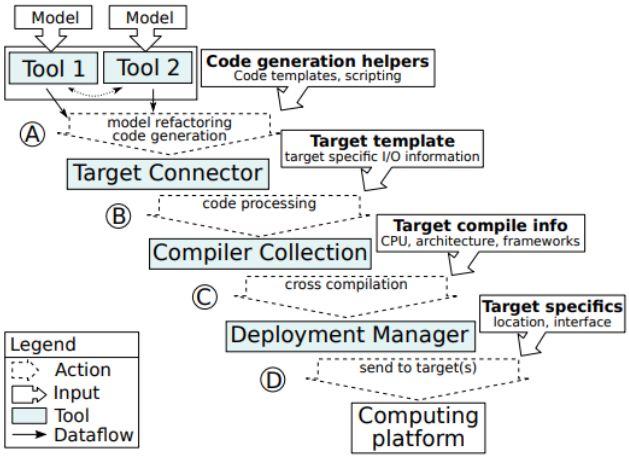

Рисунок 2.5 - Огляд необхідних інструментів для розробки програмного забезпечення для вбудованого керування

4. Останній крок – надсилання виконуваного файлу на обчислювальну платформу. Для цього менеджеру розгортання потрібні специфічні дані про ціль, такі як розташування фактичної цілі, щоб мати змогу підключитися до неї, або необхідний інтерфейс зв'язку. Типові функції менеджера розгортання також включають: віддалений запуск виконання програмного забезпечення на обчислювальній платформі або надсилання інформації журналу назад на платформу розробки.

Кроки від 1 до 4 зазвичай обробляються одним інструментом розгортання або технічно набором інструментів, оскільки розгортання, ймовірно, виконується кількома спеціалізованими інструментами [36]. Ці інструменти мають доступ до інформації про ресурси та можливості цільової платформи та можуть надавати ці дані програмному забезпеченню. Крос-компіляція програмного забезпечення також залежить від цільової платформи, тому набір інструментів розгортання надає правильні інструменти крос-компіляції на кросі (C). Надсилання скомпільованого виконуваного файлу на цільову платформу вимагає підтримки на самій цільовій платформі, оскільки виконуваний файл має бути прийнятий, збережений та виконаний на цільовій платформі. Ця підтримка на цільовій платформі повинна відповідати очікуваному протоколу зв'язку. Тому вона також забезпечується набором інструментів розгортання і зазвичай потребує одноразо-

вого встановлення на цільовій платформі. Набір інструментів розгортання в поєднанні з підтримкою на цільовій платформі також надає засоби для надсилання журналів та інформації про помилки назад на платформу розробки.

Прикладом такого набору інструментів розгортання є 20-sim 4C, який включає описані кроки та вимоги. Його цільове призначення - забезпечити цю підтримку для 20-sim, але 20-sim 4C також можна використовувати з іншими наборами інструментів моделювання [37]. У таких ситуаціях (згенерований) вихідний код повинен супроводжуватися додатковими файлами конфігурації. Деталі конфігурації, надані цими файлами, використовуються 20-sim 4C для крос-компіляції та розгортання програмного забезпечення.

2.3 Інструменти тестування, верифікації та аналізу покриття програмного забезпеченняту

У попередньому розділі пояснювалися різні методології, що є частиною способу роботи з проектування програмних моделей, а також способи їх конвертації та розгортання. У цьому розділі ці методології наведено на прикладі реальних інструментів будь-якого набору інструментів, наприклад, набору інструментів TERRA і показано, які функції повинні надавати інструменти для підтримки описаного способу роботи. Також обговорювалися додаткові ідеї та можливості інструментарію.

Графічне моделювання

Найважливішою та центральною частиною способу роботи є моделі, оскільки всі кроки проектування обертаються навколо них. Графічні моделі та інструменти моделювання ще більше покращують розуміння та зручність обслуговування моделей і способу роботи. Сучасні інструменти графічного моделювання, які називаються *редакторами,* ймовірно, інтерактивно направляють користувача до використання правильного синтаксису та семантики моделі, вказують на помилки та допомагають різними іншими способами під час процесу проектування [39]. Тим самим підвищується продуктивність дизайнера, оскільки

дизайнер може безпосередньо виправити проблему та навчитися на ній, тому такого типу проблем можна запобігти наступного разу.

Першим кроком у роботі над вбудованим програмним забезпеченням для кіберфізичної системи є розробка архітектурної моделі. Така модель забезпечує загальний огляд і повинна базуватися на методології CPC, оскільки архітектурна модель зазвичай вимагає компонентів, портів та з'єднань.

Метамодель CPC показано на рисунку 3.6, яка виступає базовою реалізацією інших метамоделей. Ці інші метамоделі розширюють метамодель CPC, як показано відкритими стрілками на рисунку, додаючи більше специфічної семантики моделі. Використовуючи ту саму базову реалізацію, моделі стають (більш) сумісними одна з одною, що є однією з причин використання методологій CPC [38]. Крім того, це економить ресурси розробки, оскільки цю базу та її підтримку потрібно реалізувати лише один раз, а потім їх можна використовувати повторно для всіх інших спеціалізованих метамоделей та їхніх інструментів.

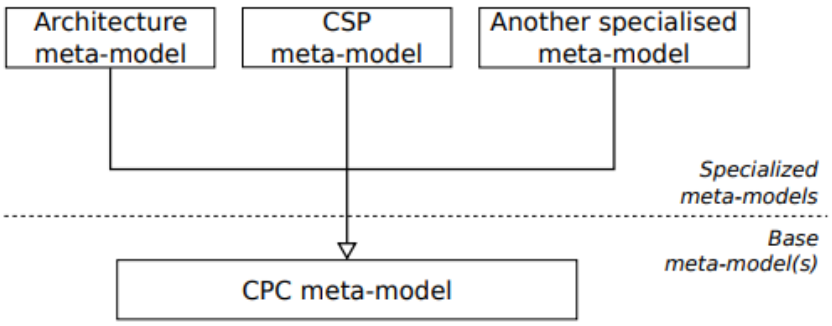

Рисунок 2.6 - Діаграма успадкування, що показує спільну модульність мета моделей

За бажанням, але не показано на рисунку, метамодель може повторно використовувати кілька базових або спеціалізованих метамоделей модульним способом. Наприклад, метамодель архітектури може повторно використовувати метамодель CSP, оскільки вона визначає визначення для уточнення зв'язку між процесами та компонентами в архітектурній метамоделі, що може покращити гнучкість (і складність) можливостей моделювання архітектурних моделей. У

випадку, якщо метамодель архітектури фактично реалізує метамодель CSP , вона зробить пов'язані з CSP функції, такі як формальна перевірка, доступними для архітектурних моделей та інструментів без особливих зусиль.

Інструмент архітектурного моделювання, або *редактор архітектури,* надає засоби для моделювання архітектури програмного забезпечення системи шляхом визначення (керуючих) компонентів (крок (1)). Під час кроків о та О ці програмні компоненти надаються з їх фактичною реалізацією. Їх реалізації повинні базуватися на метамоделі, яка розширює метамодель CPC. Крім того, *інтерфейс компонента та інтерфейс реалізації* компонента повинні бути сумісними. Обидва типи інтерфейсів мають свою власну роль у синхронізації компонента архітектури та його реалізації:

- Модель архітектури визначає інтерфейс кожного компонента, який вказує, які порти доступні або необхідні.
- Реалізація компонента повинна відповідати цьому інтерфейсу, надаючи ті самі порти (зі зворотним напрямком), щоб потік зв'язку міг переходити від архітектурної моделі до правильного місця в моделях реалізації.

Крім того, оскільки ці моделі реалізації повинні бути легко пов'язані з архітектурними моделями, їхні метамоделі також повинні підтримувати методологію CPC.

Спеціалізована метамодель не потрібна в ситуаціях, коли архітектурні компоненти не потребують реалізації, розробленої за допомогою набору інструментів моделювання [40]. Це , наприклад, той випадок, коли реалізація безпосередньо розроблена мовою програмування або коли реалізація забезпечується зовнішнім інструментом. Інструмент моделювання архітектури повинен забезпечувати пряму підтримку для цих випадків. Це може, наприклад, включати надання текстового редактора або засобів для зміни параметрів моделі для точного налаштування зовнішньої моделі.

Генерація коду

Інструменти підтримки генерації коду перетворюють моделі на вихідний код за допомогою перетворень моделі в код, також відомих як генерація коду.

Цей вихідний код разом із фреймворком(ами) можна скомпілювати у фактичний виконуваний застосунок, як показано на рисунку 2.7. У певному сенсі, інструменти генерації коду додають інформацію, необхідну для виконання моделі.

Для належної компіляції вихідного коду у виконуваний код використовується один або декілька фреймворків. Ці фреймворки містять *статичний код* , який є незалежним від моделі для забезпечення певної функціональності, такої як механізм виконання моделей. Інструменти генерації коду використовують ці фреймворки для зменшення генерації статичного коду, що підвищує зручність підтримки вихідного коду та самого інструменту генерації коду.

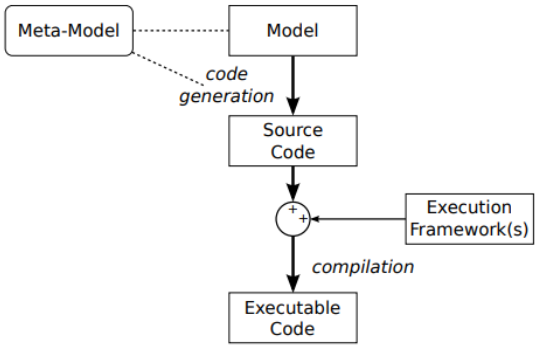

Рисунок 2.7- Огляд необхідних кроків для перетворення моделі на виконуваний за стосунок

LUNA є прикладом такої системи виконання. Кожна метамодель зазвичай підтримується супутнім механізмом виконання для підтримки згенерованого коду. LUNA надає, наприклад, механізм виконання CSP, який містить статичний код для підтримки конструкцій CSP, планувальників та каналів зустрічей.

Інші типи фреймворків, що містять, наприклад, розширені або складні датчики, зазвичай використовуються під час створення коду для кіберфізичних систем. Ці додаткові фреймворки потім використовуються для забезпечення статичного коду або драйверів для цих складних апаратних компонентів. ROS є, наприклад, типовим додатковим фреймворком, він підтримує всі види складних датчиків, таких як далекоміри, камери, датчики сили, крутного моменту та дотику.

Фреймворк також може забезпечувати підтримку для специфічних,

складних завдань, пов'язаних з контролерами, таких як декартові або кінематичні контролери. Генерація коду таких фреймворків вимагає спеціалізації метамоделей, здатних вирішувати ці складні завдання, що потребує спеціалізованої інструментальної підтримки. Orocos – це фреймворк, який забезпечує підтримку для таких завдань.

Інструменти генерації коду також потрібні, коли зовнішній інструмент не надає явної метамоделі. Замість використання цільової метамоделі для перетворення вихідної моделі, інструмент генерації коду повинен надати власну версію цільової семантики для генерації цільового коду. Зазвичай це більш схильне до помилок, оскільки цільова семантика не повністю інтерпретується правильно, або є невідомою та потребує зворотного проектування (вгадування), або була змінена з часом, а семантика, що використовується інструментом, не була відповідно оновлена. Наприклад, інструмент формальної перевірки FDR2 використовує звичайний текстовий файл як вхідні дані, використовуючи машинозчитуваний синтаксис CSP. Перетворення моделі CSP у такий файл вимагає перетворення моделі в текст, щоб модель CSP могла зчитуватися зовнішнім інструментом, щоб формально перевірити оригінальну модель CSP.

Перетворення моделі в модель

Перетворення моделі в модель є надійнішим методом перетворення порівняно з перетвореннями моделі в код, описаними в попередньому розділі. Вони використовують суворо визначену структуру моделі метамоделей вихідної та цільової моделей. Якщо вихідна або цільова метамодель змінюється, інструмент можна перекомпілювати, і він також використовує оновлену метамодель. У ситуаціях, коли метамодель зазнала деяких різких змін, перекомпіляція не вдається, і розробнику інструменту необхідно врахувати ці зміни. У всіх ситуаціях оновлений інструмент використовує найновішу семантику моделі або просто більше не підтримує найновішу версію метамоделі. Зрештою, інструмент ніколи не намагатиметься (і не повинен) використовувати застарілу семантику для своїх перетворень.

Крім того, для перетворень моделі до моделі можливо використовувати

одну й ту саму метамодель як для вихідної, так і для цільової моделі. Це, наприклад, випадок, коли перетворення використовується для оптимізації моделі для певних вимог, таких як оптимальна швидкість виконання моделі або низьке використання ресурсів. У таких випадках вся надлишкова інформація про модель, визначена цілями оптимізації, видаляється без впливу на передбачувану поведінку моделі. Отримана модель може виглядати зовсім інакше порівняно з вихідною моделлю та може більше не відповідати точці зору розробника моделювання.

Оскільки результат оптимізації все ще використовує ту саму цільову модель, усі інші інструменти все ще можуть використовуватися для виконання своїх завдань на оптимізованій моделі, тому оптимізацію моделі можна розглядати та використовувати як проміжний крок у ланцюжку інструментів моделювання. Цей тип перетворення моделі зазвичай виконується перед перетвореннями моделі в код, щоб автоматично генерувати оптимальну реалізацію моделі для виконання на цільовій платформі.

Спільне моделювання

Як пояснювалося раніше, для спільного моделювання потрібен інструмент, який керує моделюванням. Цей інструмент повинен мати знання про архітектурну структуру моделі, щоб мати змогу запустити моделювання.

Додатковою та бажаною особливістю інструменту спільного моделювання є можливість вибору конкретної реалізації для кожного визначеного інтерфейсу компонента. Наприклад, компоненти, що забезпечують зв'язок з фактичною кіберфізичною системою, слід моделювати за допомогою реалізації компонента, яка моделює апаратне забезпечення або здатна підключатися до динамічної моделі установки. Це зображено компонентом-заглушкою вводу/виводу на рисунку 2.2, який замінюється реальною реалізацією вводу/виводу на пізнішому етапі. Це запобігає необхідності завершення та доступності фактичної системи та вирішує проблеми безпеки та будівництва на ранніх етапах розробки системи. На пізніших етапах розробки системи вибрані реалізації компонентів можуть забезпечити підтримку моделювання апаратного забезпечення в циклі.

Цю функцію також можна використовувати для випробування різних ідей реалізації компонентів. Особливо в поєднанні з *інструментом керування моделями,* який здатний відстежувати різні реалізації компонентів та складати узгоджені набори реалізацій компонентів архітектурної моделі. Ці набори реалізацій компонентів також можна використовувати для моделювання певних *сценаріїв.*

Компоненти загальної архітектури

Зроблено висновок про бажаність використання універсального архітектурного компонента (GAC). Такий компонент забезпечує реалізацію універсального компонента, щоб запобігти марнуванню ресурсів проектування через повторне винаходження такої структури компонентів знову і знову. GAC повинен бути придатним для використання в різних ситуаціях, щоб зробити його зручним методом моделювання з урахуванням способу роботи. Тому він повинен надавати функції, необхідні для всіх (або принаймні більшості) реалізацій компонентів , але таким чином, щоб ці функції були необов'язковими та не обмежували проектувальників у їхніх бажаних способах роботи.

Найважливішим є керування *життєвим циклом* компонента під час його виконання. Усі компоненти необхідно ініціалізувати, налаштувати, запустити, виконати та зупинити під час виконання програми. Підтримка такого життєвого циклу може бути забезпечена реалізацією кінцевого автомата (FSM). Командний інтерфейс буде потрібен для забезпечення засобів зовнішнього впливу на стани життєвого циклу компонента.

Кожен компонент також потребує певної форми підтримки безпеки. GAC повинен забезпечувати механізми для перевірки правильності вхідних та вихідних сигналів. Наприклад, може знадобитися, щоб ці сигнали залишалися в межах відповідних очікуваних діапазонів. Якщо це не так, сигнал помилки необхідно поширити на (локальні) обробники помилок. Ці обробники помилок здатні виправляти поведінку кіберфізичної системи або (частково) зупиняти її безпечним способом, щоб запобігти пошкодженню самої системи або її середовища. GAC має бути повторно використаним двома способами:

- Він використовується в різних реалізаціях компонентів та ситуаціях.

GAC повинен забезпечити базову реалізацію, сумісну з (більшістю) цих можливих випадків використання.

- Фактична реалізація компонентів за допомогою GAC підтримує складну частину кіберфізичної системи, інакше використання GAC не мало б сенсу. Проектування та тестування таких складних компонентів потребує часу та зусиль, тому згодом вони повинні бути придатними для використання в різних (частинах) кіберфізичних систем.

Вимоги до GAC, призводять до *інтерфейсу компонентів GAC*, який визначає сигнали для команд, даних датчиків та виконавчих механізмів тощо. Інтерфейси різних GAC повинні бути з'єднувальними, щоб можна було побудувати мережу компонентів, що призведе до створення архітектури програмного забезпечення для програмного забезпечення керування кіберфізичною системою.

Структура виконання

Платформа виконання розташована між інструментами моделювання та цільовою платформою, як показано на рисунку 2.7. Вона надає *інтерфейс прикладного програмування* (API) та відповідну реалізацію, забезпечуючи достатню підтримку для мінімізації обсягу згенерованого коду. В результаті інструменти генерації коду не потребують щоразу генерувати необхідний *статичний код.* Відповідна платформа виконання також повинна забезпечувати підтримку цільової платформи, на якій виконуватиметься програмне забезпечення керування. Платформа виконання повинна забезпечувати *рівень абстракції* між наданим інтерфейсом та фактичним обладнанням і його можливостями. Вона реалізує вимоги для правильного використання обладнання, такого як підключені периферійні пристрої, датчики та виконавчі механізми, які потім можуть використовуватися іншими компонентами платформи або самими програмами. Крім того, бажано підтримувати реалізацію і для платформи розробки. Це забезпечить засоби для легшого виконання та тестування програмного забезпечення в середовищі, в якому працює розробник, що призведе до легкого тестування, навіть коли цільова платформа ще недоступна.

Структура виконання повинна гарантувати, що терміни завжди

дотримуються в межах визначеного часу, що називається підтримкою жорсткого реального часу. Жорсткі терміни реального часу необхідні для належного функціонування реалізації законів керування циклом.

Метод роботи, який підходить для проектування (вбудованого) ПЗ керування для кіберфізичних систем з використанням методів моделювання. Він інтегрований та сумісний з траєкторіями розвитку електричної та механічної областей. Крім того, метод роботи розроблений таким чином, щоб бути максимально масштабованим. Наприклад, він надає та описує фази моделювання, які можна використовувати для складних кіберфізичних систем. З іншого боку, всі кроки є необов'язковими, тому прості розробки програмного забезпечення не вимагають реалізації всіх етапів проектування. Без реалізації цих вимог спосіб роботи втрачає значну частину своєї цінності.

2.4 Висновок по розділу

У другому розділі було досліджено методи та інструменти розробки програмного забезпечення кіберфізичних систем. Розглянуто підходи до проєктування програмного забезпечення вбудованих систем керування, що забезпечують стабільне функціонування апаратно-програмних комплексів. Проаналізовано методи моделювання архітектури кіберфізичних систем, які дозволяють оптимізувати структуру програмних компонентів та взаємодію між ними. Окрему увагу приділено інструментам тестування, верифікації та аналізу покриття програмного забезпечення, що сприяють підвищенню надійності та безпеки систем. У результаті визначено ефективні технологічні підходи до створення сучасних кіберфізичних рішень.

РОЗДІЛ 3. РОЗРОБКА ТА ОЦІНКА ЕФЕКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КІБЕРФІЗИЧНОЇ СИСТЕМИ

3.1 Проєктування загальної архітектури кіберфізичної системи

Інформацію про проектування та реалізацію *компонента загальної архітектури* (GAC). Такий компонент архітектури забезпечує базовий шаблон для проектування реалізації компонентів кіберфізичних систем. Це частина способу роботи, але ідеї GAC також можна використовувати в інших ситуаціях.

Компонент загальної архітектури має такі властивості:

- Загальність

Цей компонент можна використовувати для всіх видів (кіберфізичних) систем, починаючи від низькоресурсних вбудованих систем і закінчуючи великими промисловими або медичними системами. Крім того, GAC повинен підтримувати всі види завдань, починаючи від низькорівневих контролерів циклу і закінчуючи алгоритмами керування, які планують поведінку системи. Іншими словами: компонент має бути *універсальним.*

- Архітектурний

Цей компонент придатний для використання як частина архітектури системи. Архітектура системи складається з компонентів, які разом здатні керувати фізичною системою. Ці компоненти пов'язані один з одним, щоб мати змогу разом працювати над виконанням необхідних завдань. GA C має бути частиною цієї *архітектури* компонентів та забезпечувати підтримку для цього.

- Компонент

GAC поводиться як будь-який інший програмний компонент, він забезпечує базову підтримку для часто необхідних функцій.

GAC, про який йдеться в цьому розділі, є кресленням або шаблоном з базовими функціональними можливостями/підтримкою для побудови реальних реалізацій компонентів, що походять від цього шаблонного GAC, а не реальної реалізації компонента. Чертеж GAC у цьому розділі називається *шаблонним GAC* ,

а реалізація компонента на основі GAC називається *спеціалізованим GAC*. Зауважте, що за замовчуванням мається на увазі шаблонний GAC, коли ні шаблонний GAC, ні спеціалізований GAC явно не згадуються.

GAC повинен підтримувати всі види компонентів, які можна використовувати в кіберфізичних системах, починаючи від гуманоїдних і закінчуючи медичними та промисловими роботами, оскільки він має бути універсальним. Цей діапазон цільових систем потребує спеціальної підтримки, яка обговорюється в першому розділі цього розділу. Доступні моделі компонентів та супутні фреймворки, які можна використовувати для реалізації GAC, обговорюються в наступному розділі. Далі описується підхід до проектування та реалізації GAC. Після цього наводиться приклад використання, який показує, що GAC можна використовувати для реалізації програмного забезпечення керування для реальної кіберфізичної системи. Розділ завершується обговоренням проектування, реалізації та прикладу використання GAC.

Вимоги

Вимоги до GAC, що базуються на властивостях, описаних вище, та на необхідних функціях GAC, детально розглянуті в цьому розділі. Вимоги до способу роботи зосереджені або на можливості повторного використання та гнучкості, тобто на вимогах, які гарантують, що GAC дійсно є універсальним, або на необхідних функціях, що визначають його поведінку та функціональність.

Вимога 1: GAC має бути багаторазовим

Компонент загальної архітектури має бути універсальним; в ідеалі він має бути придатним для використання в усіх видах реалізації кіберфізичних систем. Тому як шаблонний GAC, так і спеціалізований GAC повинні бути придатними для повторного використання.

Шаблон GAC повинен бути застосовним до різних ситуацій та вимог до дизайну, тому спеціалізовані GAC також стають застосовними до широкого кола ситуацій.

З іншого боку, самі спеціалізовані GAC також повинні бути придатними для повторного використання, щоб їх можна було використовувати для реалізації

кількох частин системи. Наприклад, підтримка конфігурації дозволяє точно налаштувати спеціалізований GAC контролера петлі, що дозволяє керувати як лівою, так і правою частинами симетричної системи за допомогою одного й того ж спеціалізованого GAC.

Вимога 2: GAC повинен підтримувати методологію CPC

Як зазначено методологію CPC необхідно використовувати як базовий тип метамоделі. Використовуючи метамодель CPC для шаблону моделі GAC, стає можливим легко пов'язати спеціалізовані GAC з іншими GAC (або іншими компонентами), що базуються на метамоделі CPC.

Вимога 3: GAC повинен бути здатним працювати в режимі реального часу.

ГАК повинен мати можливість працювати на різних рівнях реального часу, щоб бути придатним для використання як компонентна реалізація всіх видів сценаріїв використання. ГАК з контролером циклу керування повинні забезпечувати підтримку жорсткого реального часу для реалізації коду контролера. З іншого боку, контролери нагляду або послідовності не потребують суворих гарантій жорсткого реального часу. Однак, зв'язок між ГАК з різними рівнями реального часу є обов'язковим. Наприклад, ГАК з м'яким прийняттям рішень у реальному часі зазвичай використовуються для призначення завдань ГАК з жорстким контролем циклу керування в реальному часі.

Підтримка кількох рівнів реального часу також потрібна в самому GAC, це запобігає надмірному, зайвому використанню ресурсів (і, отже, непотрібній траті цих ресурсів). Частини GAC, пов'язані з керуванням, які повинні періодично виконувати свої обчислення, щоб запобігти нестабільній динамічній поведінці, повинні працювати з жорсткими гарантіями реального часу. Тоді як обробники команд, наприклад, задовольняються м'якими гарантіями реального часу, оскільки не має значення, чи команда інтерпретується трохи пізніше, ніж вона була видана. Потрібна гнучкість, тому ці рівні реального часу можна вибрати під час проектування спеціалізованого GAC або під час його використання для моделювання програмного забезпечення керування.

Вимога 4: GAC повинен забезпечити засоби для підтримки заходів безпеки для вирішення небажаних ситуацій.

Навіть попри те, що методи модельно-орієнтованого проектування допомагають у розробці правильного програмного забезпечення, заходи безпеки все ще необхідні для обробки небажаних ситуацій. Тому GAC повинен забезпечити засоби для обробки аспектів безпеки, таких як виявлення та обробка небезпечних ситуацій, як на локальному, так і на глобальному рівні.

*Вимога 5***:** *GAC повинен забезпечити засоби для його розширення до спеціалізованої компонентної реалізації.*

Шаблонний GAC повинен забезпечувати засоби для фактичного визначення його необхідної поведінки, і таким чином перетворювати його на спеціалізований GAC або спеціалізований компонент. Наприклад, компонент контролера циклу повинен виконувати фактичний алгоритм контролера циклу, тому GAC повинен забезпечувати засоби для додавання цього алгоритму та забезпечення його періодичного виконання.

Отже, GAC повинен надавати заповнювачі, які називаються *гачками (hooks)* . Ці гачки надають засоби для додавання фактичного вмісту реалізації або корисного навантаження компонента. Кожен гачок забезпечує підтримку певної частини реалізації, яку дизайнери використовують для формування шаблонного GAC у спеціалізовані GAC. Забезпечуючи достатню кількість таких гачків, шаблонний GAC можна використовувати для створення будь-якого бажаного компонента. Фактичні реалізації цих гачків повинні бути необов'язковими та дозволяти їх визначення в кількох форматах, щоб GAC був якомога більш універсальним.

Вимога 5.1: Гаки GAC повинні підтримувати вихідний код

Корисне навантаження хука має бути доступним для програмного забезпечення, яке створюється з вихідного коду. Навіть при використанні методів MDD моделі зазвичай перетворюються на вихідний код під час створення програми. Тому хуки повинні принаймні підтримувати вихідний код як своє корисне навантаження.

Вимога 5.2: Гачки GAC повинні підтримувати моделі

Повинна бути можливість використовувати моделі як корисне навантаження для гачка, тому спеціалізований GAC також може бути побудований за допомогою методів MDD. Ці моделі можна легко підключити до GAC, якщо вони також базуються на метамоделі CPC.

*Вимога 6: Шаблон GAC має бути формально правильним***.**

GAC використовуються в різних ситуаціях завдяки своїй багаторазовій та гнучкій природі. Вони повинні належним чином функціонувати в усіх цих ситуаціях, тому шаблон GAC повинен бути формально коректним, щоб запобігти (неочікуваним) проблемам, таким як глухі блокування, через ситуацію, в якій він використовується.

Вимога 6.1: Спеціалізований GAC повинен бути придатним для офіційної перевірки

Спеціалізований GAC також повинен мати можливість формальної перевірки. Впровадження шаблону GAC у спеціалізований GAC може призвести до нових проблем, які можна виявити за допомогою формальної перевірки. Крім того, якщо спеціалізований GAC є формально перевіреним, стає можливим формально перевірити повну мережу спеціалізованих GAC або все програмне забезпечення керування. Це допомагає знаходити помилки проектування, що виникають через неправильно додані зв'язки між спеціалізованими GAC.

Вимога 7: GAC має бути придатним для використання в ієрархічних мережевих ситуаціях.

Повинна бути можливість використовувати GAC в мережі інших GAC, в якій вони разом утворюють керуюче програмне забезпечення. Крім того, один або декілька GAC повинні бути здатні забезпечувати корисне навантаження іншого GAC. За цієї вимоги складний GAC можна розділити на кілька GAC, які разом забезпечують складну поведінку. Ці прості GAC потім можна (повторно) використовувати для різних ситуацій.

Існуючі моделі компонентів

Як зазначено, доступні кілька реалізацій кіберфізичних програмних

фреймворків, таких як Orocos та ROS. Більшість із них використовують певну модель власного компонента, інтерфейс або шаблон, які має реалізувати інженер. Як згадувалося раніше, ця робота не є достатньо гнучкою та багаторазовою для розробки програмного забезпечення керування для широкого спектру кіберфізичних систем, але їхні ідеї проектування враховуються.

Модель *компонентів БРІКС* (BCM) пропонує підхід *5C*. Вона визначає наступні 5 завдань (5C): обчислення, комунікація, координація, конфігурація та композиція. Як згадувалося раніше, у цій роботі використовується розділення завдань, тому для проектування GAC використовується реалізація у формі 5C.

Orocos – це фреймворк, орієнтований на кіберфізичну розробку, зосереджуючись, зокрема, на складніших компонентах, таких як послідовність та компоненти диспетчерського керування.

Orocos , показаний на рисунку 3.1, забезпечує основу для компонента Orocos. Компонент пропонує сервіси через інтерфейс операцій та запитує їх через виклики операцій (OperationCallers). Порти потоку використовуються як механізм передачі даних між компонентами. У центрі компонента знаходиться його механізм виконання (Execution Engine) обробляє виконання компонента, зображеного шестернями на задньому плані рисунка. Кожен компонент має кінцевий автомат, який обробляє потік станів компонента. Користувач може додавати функціональність до перехоплювачів, які активуються при переходах між станами. Orocos надає середовище для написання сценаріїв у реальному часі для програмування компонента без зміни його вихідного коду , що дозволяє швидкі, ітеративні цикли проектування. Це лише підсумовує основні характеристики компонентів Orocos, насправді це досить складний компонент з безліччю інших функцій та можливостей, що робить його дуже гнучким для будь-яких ситуацій та вимог.

Використання моделі компонентів Orocos призводить до частково реального часу програмного забезпечення: сам компонент працює в реальному часі, тобто під час фактичного виконання використовуються лише детерміновані операції. Але компоненти (по суті, їхні потоки ОС) плануються операційною

системою (ОС), що ускладнює забезпечення жорстких гарантій планування компонентів у реальному часі. На жаль, поведінка планування сильно залежить від реалізації операційної системи (реального часу).

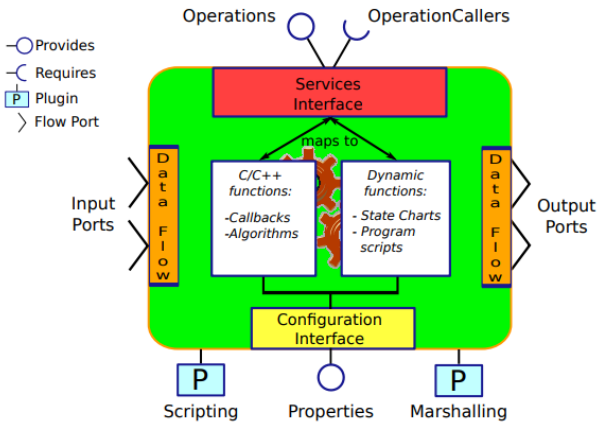

Рисунок 3.1 - Модель компонентів Orocos

Занадто потужне обладнання призводить до неправильного виконання програмних компонентів реального часу, таких як компоненти послідовності або диспетчерського керування. Однак це рішення не підходить для вбудованих обчислювальних платформ, тому потрібна ОС реального часу.

ROS – це ще одна відома та широко використовувана програмна платформа для керування роботами. Основна увага ROS приділяється підтримці повторного використання коду в дослідженнях та розробках робототехніки.

Він забезпечує високорівневі (керівні) рішення для передачі повідомлень між процесами та управління пакетами, надаючи механізм підписки видавця. На жаль, ROS не надає певної моделі компонентів, не може гарантувати роботу в режимі реального часу та працює лише на ПК загального призначення та операційних системах, а також поки що не підтримує вбудовані платформи.

Хоча ROS не підтримує роботу в жорсткому реальному часі, це гарна платформа для забезпечення сумісності завдяки кількості підтримуваних (спеціалізованих) драйверів для датчиків та виконавчих механізмів. Ці драйвери можна використовувати з гарантіями роботи в м'якому реальному часі, щоб забезпечити інші компоненти своїми даними. Це особливо можливо для складних датчиків,

таких як камери або лазерні далекоміри, які зазвичай використовуються компонентами послідовного або диспетчерського керування, що самі працюють у м'якому реальному часі.

Дизайн

Шаблонний дизайн GAC залишається простим і легким, щоб забезпечити його використання на вбудованих платформах. Завдяки використанню методів MDD, змодельований GAC є платформонезалежним і його легко підключати до решти моделей керуючого програмного забезпечення. Ці варіанти проектування є першим кроком до розробки GAC багаторазового використання, як того вимагає Вимога 1.

Огляд конструкції GAC показано на рисунку 3.2. Його функціональність розподілена на 4 блоки, які працюють разом та забезпечують реалізацію GAC. Крім того, GAC розділений на три рівні виконання з власними властивостями:

- Шар одноразового виконання

Блоки в цьому шарі виконуються один раз, під час ініціалізації GAC.

- Синхронний шар

Блоки в цьому шарі виконуються періодично.

- Асинхронний шар

Блоки цього шару виконуються, коли надходить подія, яку потрібно обробити.

Блоки GAC розміщуються в одному або кількох із цих шарів, залежно від їхніх вимог.

Існує кілька потоків зв'язку між блоками, а також між блоками всередині GAC та від блоків до зовнішнього боку GAC. Ці потоки зв'язку поділяються на три типи. Два з цих типів попередньо визначені шаблоном GAC, що складаються з координаційного зв'язку та зв'язку про помилки. Третій тип представляє додаткові потоки зв'язку, що визначаються користувачем, як це вимагається для спеціалізованих GAC, тобто зв'язок вводу/виводу від/до зовнішнього боку компонента.

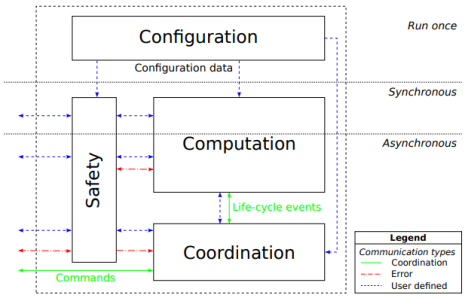

Рисунок 3.2 - Огляд проекту компонента загальної архітектури

Більшість вхідних комунікаційних потоків з-за меж GAC проходять через *блок безпеки* до місця призначення. Вхідні дані з-за меж GAC вважаються небезпечними, блок безпеки перевіряє дані на наявність помилок, неочікуваних або небажаних значень та обробляє ці сигнали за необхідності. Дані, що залишають блок безпеки, вважаються безпечними. Дані, що залишають GAC, також проходять через блок безпеки, у цьому випадку блок також перевіряє, чи є дані правильними, наприклад, чи знаходяться в очікуваному або дозволеному діапазоні. Якщо це не так, це може свідчити про те, що GAC несправний і потребує уваги для вирішення проблеми.

Детальна інформація про розподіл обов'язків у рамках GAC наведена в наступному розділі. Кожен із цих розподілів та те, як він впливає на структуру GAC, обговорюється в наступних розділах.

Розділення турбот

Як показано на рисунку 3.2, структура GAC розділена на 4 окремі блоки, кожен з яких обробляє окрему, специфічну частину GAC. Три з блоків безпосередньо походять від підходу 5C моделі компонентів БРІКС. Хоча питання комунікації та композиції безпосередньо не видно на рисунку, вони також впливають на GAC.

5C використовуються в дизайні GAC наступним чином:

- Блок *обчислень* GAC забезпечує засоби для виконання (керуючого) алгоритму, тобто корисного навантаження GAC.
- Блок *координації* керує машиною станів життєвого циклу для синхронізації дій або станів компонента.

- *Блок конфігурації* GAC надає значення параметрів спеціалізованих компонентів, що збільшує можливість повторного використання шаблонних GAC та спеціалізованих GAC.
- *Підрозділ комунікації* піклується про зв'язок, додаючи порти до компонента, тим самим забезпечуючи зв'язок із середовищем компонента.
- Питання *композиції* реалізується опосередковано за допомогою різних можливостей створення архітектурної мережі глобальних акційних центрів (GAC).

Безпека не входить до 5 початкових питань, але її роль достатньо важлива, щоб її включити на чільне місце в дизайні GAC. Оскільки на цьому чільному місці дизайнер, ймовірно, помітить блок і використає його, що призведе до створення спеціалізованих GAC з високою (вищою) якістю.

Ролі в GAC кожного з цих питань (включаючи безпеку) детальніше обговорюються в наступних розділах.

Обчислення

Блок обчислень відповідає за обчислення корисного навантаження GAC. Він повинен забезпечувати засоби для додавання визначених користувачем обчислювальних перехоплювачів, по одному для кожного стану життєвого циклу, як того вимагає Вимога 5. Реалізація цих обчислювальних перехоплювачів може бути надана розробником у вигляді моделей на основі CPC або у вигляді прямого вихідного коду, як це визначено вимогами 5.1 та 5.2. Реалізація перехоплювача, що належить до активного стану, повинна виконуватися блоком координації.

Також можливо використовувати інші GAC як (часткову) реалізацію блоку обчислення, це пояснюється далі, коли пояснюється питання композиції. Блок обчислення розміщується як на синхронному, так і на асинхронному рівнях, щоб зробити обчислення як подієвими, так і періодичними.

Координація

Блок координації відповідає за *життєвий цикл* та користувацькі автомати станів GAC. Автомат життєвого циклу використовується для забезпечення звичайного життєвого циклу GAC, тоді як користувацький автомат станів

дозволяє налаштовувати, більш детально визначати стани компонентів.

Блок здатний отримувати команди, які по суті є подіями, від інших GAC для керування активним станом життєвого циклу GAC. Наприклад, GAC планування завдань відстежує довгострокові цілі системи та керує іншими GAC для досягнення цих цілей за допомогою подій. Зміни активного стану передаються обчислювальному блоку, який відповідно змінює свою поведінку, використовуючи правильний перехоплювач для активного стану. Блок координації розміщено на асинхронному рівні виконання, оскільки він повністю базується на подіях.

Життєвий цикл кінцевого автомата, що існує в блоці координації, показано на рисунку 3.3. Назви подій написані великими літерами, а назви станів – жирним шрифтом, як на рисунку, так і в тексті. Рисунок спрощений і має лише кілька станів і переходів.

ГАК запускається у *стані Init.* Цей стан призначений для виконання ініціалізації ГАК та дій повернення механічної системи до початкового положення. Коли ініціалізація завершена та видається *подія INIT_READY , ГАК переходить у стан InitReady.* Цей стан використовується для синхронізації всіх ГАК: після того, як усі завершили свою ініціалізацію, керівний компонент надсилає *подію RUN для переходу ГАК у стан виконання* . У цьому стані активується фактична функціональність ГАК. Зі стану виконання можна вийти або звичайним способом, зупинивши ГАК за допомогою *події STOP,* або через невиправну помилку, про яку сигналізує *подія ERROR.* Залежно від події, переходить у *стан Stop* або *Error* відповідно. Обидва стани мають подію готовності: *STOP_READY* та *ERROR_READY.* Коли дії фіналізації, необхідні для належної зупинки ГАК, завершені, надсилається відповідна подія.

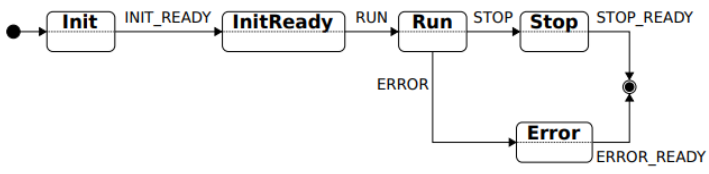

Рисунок 3.3 - Діаграма кінцевого автомата життєвого циклу GAC

Цей простий кінцевий автомат не забезпечує засобів для перезапуску GAC, коли він зупинений або виникає невиправна помилка, вихід зі стану «Виконано» відбувається лише тоді, коли програмне забезпечення завершує роботу. Перезапуск компонента небажаний, оскільки немає сенсу зупиняти компонент, поки він (активно) керує механічною частиною кіберфізичної системи. Це також не має сенсу з точки зору програмного забезпечення: архітектурна мережа компонентів раптово (тимчасово) стає неповною, що призводить до проблем, коли потрібна взаємодія з зупиненим компонентом. Очевидною причиною зупинки компонента є зміна його поведінки, наприклад, заміна його на інший компонент керування циклом через зміну активного завдання кіберфізичної системи. Рекомендується реалізувати такі ситуації шляхом впровадження користувацького кінцевого автомата та використання його станів для вибору правильного закону керування для конкретної ситуації. Це запобігає зупинці компонента, а отже, зупинці для керування механічним компонентом кіберфізичної системи. Крім того, це дозволяє максимально знизити складність реалізації GAC.

Як уже згадувалося, блок координації підтримує, окрім кінцевого автомата життєвого циклу, користувацький кінцевий автомат. Користувацький кінцевий автомат може використовуватися спеціалізованими глобальними акселераторами (GAC) для додавання користувацьких станів для більш детального контролю життєвих циклів GAC.

Конфігурація

Блок конфігурації забезпечує засоби для підтримки параметрів конфігурації спеціалізованого GAC. Значення цих параметрів надаються, коли спеціалізований GAC використовується як компонентна реалізація. Наразі параметри встановлюються лише під час ініціалізації GAC, тому блок розміщується на рівні одноразового виконання, це робиться для того, щоб зробити GAC якомога простішим.

Додаткові засоби для підтримки конфігурації під час виконання можна забезпечити, перемістивши блок на асинхронний рівень та додавши зв'язок на основі команд. Це дає змогу надсилати команди до блоку для переналаштування

глобального допоміжного центру (GAC) за потреби. Можна навіть вбудовувати значення параметрів конфігурації в команду, щоб динамічно змінювати параметри. Це може бути зручно для GAC, що містять під-GAC, які потребують налаштування.

Зв'язок

Узагальнені порти необхідно ввести в GAC для задоволення потреб комунікації. Вони є частиною парадигми CPC, яка, отже, потрібна як інструмент моделювання для проектування GAC. Порти формують інтерфейс кожного компонента та визначають своєрідний протокол зв'язку. Цей протокол «визначає», які дані будуть доступні або потрібні кожному порту. Розробник використовує цей протокол, з'єднуючи сумісні порти один з одним.

Порти різних компонентів необхідно з'єднати за допомогою каналів, щоб дані можна було передавати з одного порту на інший. Ці канали визначають потоки даних між портами компонентів. Використовуючи відповідну реалізацію каналу для кожної пари з'єднаних портів, GAC може взаємодіяти з іншими компонентами, що є частиною архітектури системи. Прикладами доступних реалізацій зв'язку є канали зустрічей та буферизовані канали.

Інструмент MDD повинен допомагати у виборі правильної реалізації каналу. Використовуючи інформацію про змодельовану архітектуру системи, він може визначити правильну або оптимальну реалізацію каналу.

Склад

Питання композиції забезпечує підтримку використання глобального акселератора (GAC) у мережі GAC. GAC можуть розташовуватися поруч один з одним, забезпечуючи *архітектурну мережу* для програмного забезпечення , або вони можуть використовуватися в *ієрархічній мережі, де* один або декілька під-GAC забезпечують функціональність батьківських GAC. Більш детальна інформація про ці можливості наведена в цьому розділі.

Приклад мережевих GAC показано на рисунку 3.4. Вихід профілю руху GAC – це потік періодично оновлюваних заданих значень, який підключено до входу PID GAC. PID GAC використовує ці задані значення для обчислення свого

керуючого сигналу, який використовується для керування його виконавчим механізмом. Зауважте, що цей приклад занадто простий, у реальній ситуації немає сенсу розділяти цю функціональність на два GAC.

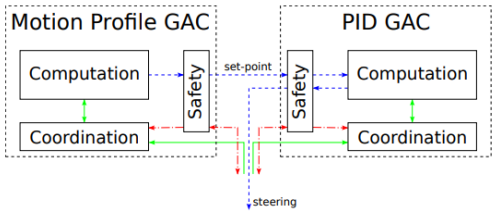

Рисунок 3.4 - Приклад мережі з двох GAC

Не має значення, чи найновіша уставка доступна ПІД-контролеру постійно. Хоча вони потребують частого оновлення, достатньо забезпечити профіль руху з м'якими гарантіями реального часу. ПІД-контролер безпосередньо керує своїм виконавчим механізмом, ці сигнали керування потрібні кожного періоду часу, щоб запобігти проблемам з динамікою фізичної системи. Отже, ПІД -контролер вимагає жорстких гарантій реального часу. Потрібні м'які чи жорсткі гарантії реального часу, залежить від контролерів та доступності ресурсів. Мережева конфігурація GAC дозволяє використовувати різні гарантії реального часу для кожного GAC.

Зв'язок між двома GAC з різними гарантіями реального часу вимагає реалізації буферизованого каналу. Це необхідно для запобігання *зниженню пріоритету* GAC з більш суворими гарантіями реального часу. Зниження пріоритету відбувається, коли більш суворі гарантії реального часу стають менш суворими іншим компонентом, оскільки GAC зі суворими гарантіями реального часу примусово очікує даних, що надходять від іншого компонента. Буферизований канал завжди має доступні дані, хоча вони можуть бути застарілими, що відрізняється від каналу зустрічі, який вимагає готовності обох компонентів до початку зв'язку.

Обидва GAC у мережевому прикладі GAC разом забезпечують сигнал керування для виконавчого механізму. Якщо в кіберфізичній системі є кілька

виконавчих механізмів, які потребують такого контролера, має сенс створити один GAC, який забезпечує сигнал керування. Цей GAC може повторно використовувати обидва окремі GAC як свою реалізацію для комунікаційного блоку. Такий GAC показано на рисунку 3.5. Під-GAC контролера циклу GAC мають сірий фон.

Ієрархічна мережа GAC показана на рисунку: Контролер циклу GAC містить два під-GAC: GAC профілю руху та GAC PID. Реалізація обох під-GAC (позначених сірим фоном) така ж, як і в попередньому прикладі, але тепер вони підключені до блоку координації контролера циклу GAC замість контрольного GAC. Разом під-GAC забезпечують реалізацію обчислювального блоку батьківського GAC. Результатом обчислювального блоку, керуючим сигналом, що надається під-GAC PID, є вихід GAC контролера циклу.

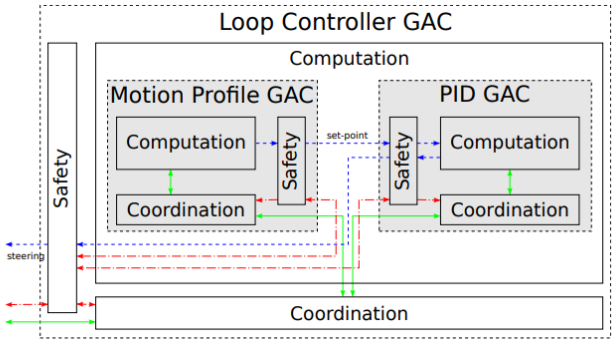

Рисунок 3.5 - Приклад налаштування ієрархічного GAC

Безпека

Блок безпеки забезпечує засоби для моніторингу потоків даних між GAC та компонентами поза ним. Як згадувалося раніше, безпека вважається важливою, тому *всі* сигнали даних проходять через цей блок безпеки. Передбачено перехоплювач для додавання користувацьких перевірок залежно від вимог спеціалізованого GAC. Наприклад, він може перевіряти, чи знаходиться значення даних у певному діапазоні, або чи значення не змінюється занадто швидко. Якщо ці користувацькі перевірки виявлять проблему, блок безпеки може змінити дані, щоб виправити її.

Якщо виправлення даних неможливе або не дозволене, блок безпеки може надіслати *подію ПОМИЛКА* до блоку координації із запитом на подальші дії. Ця подія потім може бути використана для усунення помилки на *локальному рівні.* Якщо це також неможливо, повідомляється про це диспетчерська система, і помилка обробляється на *глобальному рівні,* ймовірно, шляхом повного вимкнення кіберфізичної системи.

Сигнали про помилки, що виходять з GAC, також використовуються, коли компоненту нагляду потрібно повідомити компонент про те, що сталася глобальна помилка, і компонент потрібно вимкнути разом з рештою кіберфізичної системи. Він може перейти або в стан STOP, або в стан ERROR для вимкнення. Це залежить від поточного стану компонента та від того, чи є в ньому також помилки.

Обговорення

Порівняно з компонентом Orocos, показаним на рисунку 3.1, проект GAC має схожі блоки: потік виконання обробляється в окремому блоці, компонент Orocos має свій механізм виконання, а GAC має блок координації. Обидва обробляють координацію в окремих блоках, надаючи кілька станів життєвого циклу, які можуть бути розширені та заповнені користувачем. Конфігурація компонента обробляється обома моделями компонентів, і обидва обробляють зв'язок з іншими компонентами за допомогою узагальнених портів.

Окрім цих подібностей, конструкція GAC не надає стільки загальнодоступних інтерфейсів для налаштування компонента або виконання функціональних можливостей компонента, як компонент Orocos. GAC підтримує лише (користувацькі) команди/події для зміни активного (життєвого циклу) стану та сигнали даних, визначені користувачем, ззовні. Таке зменшення підтримки дещо обмежує гнучкість GAC, але тим самим покращує низьке використання ресурсів та значно зменшує складність GAC. Крім того, GAC забезпечує функції обробки внутрішньої безпеки, що дозволяє виявляти проблеми у вхідних та вихідних сигналах , і він здатний обробляти ці проблеми локально або глобально. Це підвищить ймовірність того, що розробник компонентів задумається над питаннями

безпеки, оскільки базова підтримка готова до заповнення деталями.

Оскільки блоки GAC суворо розділені, можливо мати кілька рівнів реального часу в межах компонентів та/або його під-GAC. Якщо розглянути ієрархічний приклад налаштування GAC, під-GAC PID повинен бути в жорсткому реальному часі, оскільки він безпосередньо керує виконавчими механізмами кіберфізичної системи, тоді як під-GAC профілю руху не потребує цих гарантій. Тому було б правдоподібно забезпечити м'які гарантії реального часу для під-GAC профілю руху, зберігаючи при цьому жорсткі гарантії реального часу для основного та PID GAC. Як згадувалося раніше, приклад занадто простий, щоб фактично розділити GAC на два під-GAC, особливо з різними обмеженнями реального часу, але для складніших ситуацій це мало б сенс.

На завершення, конструкція GAC є менш складною, ніж компонент Orocos, головним чином через обмежену доступність (складних) функцій. GAC, будучи спрощеною порівняно з компонентом Orocos, призводить до меншого використання ресурсів, що особливо зручно для (малих) вбудованих обчислювальних платформ. GAC все ще достатньо повний, щоб його можна було використовувати як загальний шаблон для компонентів кіберфізичної системи, як буде показано далі в цьому розділі. Додатковою перевагою GAC є те, що менша кількість доступних інтерфейсів призводить до менш складних конструкцій та компонентів, що призводить до меншої плутанини для (початківців) інженерів.

3.2 Реалізація програмного забезпечення та інтеграція компонентів системи

Першу фактичну реалізацію розроблено. Вона показує, що ідеї, що лежать в основі GAC, є здійсненними, хоча потрібна додаткова робота для підвищення якості та зменшення використання ресурсів, перш ніж ця реалізація стане повністю придатною для використання. Ця реалізація GAC в основному розроблена з використанням моделей CSP, тому шаблон GAC та його спеціалізовані GAC значною мірою підлягають формальній перевірці, як того вимагає Вимога 6.

Огляд цієї реалізації GAC зображено на рисунку 3.6. Рисунок базується на огляді проекту, зображеному на рисунку 3.2, але доповнений деталями реалізації, такими як описані перехоплювачі, кінцеві автомати та сигнали.

До блоку координації додано кінцевий автомат життєвого циклу. На жаль, його реалізація надається на C++ і все ще потребує конвертації в CSP, щоб зробити реалізацію GAC повністю формально перевіреною. Для ситуацій, коли потрібні додаткові стани, можна реалізувати *користувацьку перехоплювальну функцію FSM . Користувацька перехоплювальна функція*

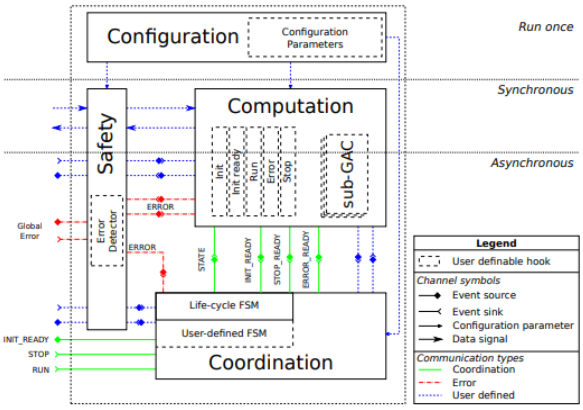

Рисунок 3.6 - Деталі реалізації компонента загальної архітектури

Каналів подій можна використовувати для надсилання додаткових користувацьких подій до користувацького автомата. Порівняно з оглядом проекту, додано подію STATE для повідомлення обчислювального блоку про зміни стану життєвого циклу, щоб він міг виконати перехоплення, що належить активному стану.

Стани життєвого циклу представлені в обчисленні, кожен стан життєвого циклу має свій власний перехоплювач. Зміни в активному стані передаються до блоку обчислення, який використовує цю інформацію для періодичного виконання правильного перехоплювача стану життєвого циклу. Події INIT_READY, STOP_READY та ERROR_READY використовуються блоком обчислення для повідомлення блоку координації про завершення відповідного стану . Наприклад, після того, як перехоплювач Init завершив свої процедури повернення до

початкової точки, йому потрібно випустити подію INIT_READY.

Блок конфігурації надає перехоплювач для реалізації визначених користувачем параметрів конфігурації. Ці параметри можна використовувати для налаштування інших перехоплювачів, що робить спеціалізований глобальний архівний центр (GAC) придатним для використання в широкому діапазоні ситуацій. Блок координації зазвичай зчитує значення параметрів з файлу конфігурації та надає їх перехоплювачу для обробки, а потім надсилає їх у відповідні місця. Як обговорювалося раніше, його можна змінити для прийняття параметрів конфігурації для налаштування GAC під час виконання.

Порівняно з оглядом глобального контролю помилок (GAC), доступні сигнали зв'язку користувача та помилок блоку безпеки наведено більш детально. Вхідні сигнали, визначені користувачем, пересилаються або до блоку координації, або до блоку обчислення. Події, пов'язані з помилками, керуються реалізацією безпеки та надсилаються, коли виникає проблема, яку не може вирішити лише блок безпеки. Канали подій ERROR до та від блоку обчислення використовуються для повідомлення під-GAC, якщо такі є, про помилки в масштабах всієї системи. Вони підключені до каналів *глобальних помилок* цих під-GAC. Перехоплювач *детектора помилок* використовується для визначення того, як потрібно перевіряти сигнали, визначені користувачем.

Реалізація GAC забезпечує всіляку додаткову підтримку, що робить GAC масштабованим та гнучким для використання в широкому діапазоні ситуацій. Однак він не надає багато приємних, але складних функцій, як компонентна модель Orocos. Цьому можна частково протидіяти шляхом розумного повторного використання реалізованих функцій у спеціалізованому GAC, як показано в наступному розділі.

Використання компонента загальної архітектури

Установка виробничої комірки, показана на рисунку 3.7, являє собою масштабовану модель ливарної машини. У цьому підрозділі вона використовується для демонстрації того, як GAC може бути використаний як основа для проектування програмного забезпечення керування реальної кіберфізичної системи.

Металеві блоки на стрічках представляють матеріали, що транспортуються через формувальну машину. Ці змодельовані матеріали мають зверху шматок заліза, тому їх можуть захоплювати електромагніти установки.

Виробнича комірка складається з 6 окремих виробничих блоків (ВБК). Стрічка подачі транспортує матеріали до зони формування, а стрічка вилучення транспортує вилучені матеріали з зони формування. Зона формування складається з трьох частин: блоку подачі, блоку формування та блоку вилучення. Блок формування складається з формувальних дверей, до яких матеріал притискається блоком подачі, імітуючи процес формування. Після цього двері відчиняються, блок вилучення захоплює відформований матеріал і поміщає його на стрічку вилучення. Для демонстрації використовується обертовий блок для підтримки потоку матеріалу. Він запобігає падінню матеріалів з стрічки вилучення, переміщуючи їх з стрічки вилучення назад на стрічку подачі, замикаючи коло.

Кожен блок керування процесом (PCU) має один або два виконавчі механізми, якими потрібно керувати. Усі вони мають двигун, який приводить у рух стрічку, переміщує роботизовану руку або відкриває дверцята формувальної машини. Блок вилучення та блок обертання мають другий виконавчий механізм у вигляді електромагніту, що дозволяє PCU захоплювати шматок матеріалу.

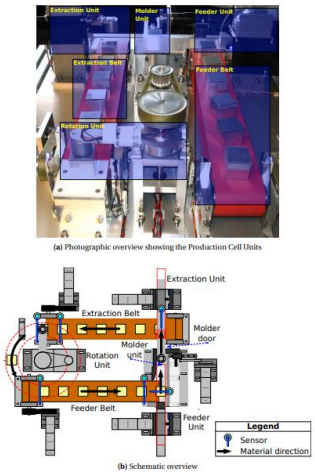

Рисунок 3.7 - Схема виробничої лінії, модель формувальної машини

Окрім виконавчих механізмів, усі PCU мають кілька датчиків. Усі дви -

гуни мають супутній обертовий енкодер для вимірювання обертання осі двигуна. Крім того, кожен PCU має один або декілька «датчиків матеріалу», що складаються з оптичних перемикачів , для виявлення того, чи зайнято певне положення в області PCU шматком матеріалу.

Два сусідні блоки розподілу матеріалів (PCU) повинні повідомити, чи можливо перемістити матеріал до наступного PCU, чи є інший шматок матеріалу на шляху.

Міркування щодо проектування PCU GAC

Через схожі вимоги до керування PCU, має сенс розробити багаторазовий ГАК PCU. Оскільки не всі PCU мають однакову кількість датчиків та виконавчих механізмів, існує компроміс між наявністю одного універсального ГАК PCU або наявності кількох спеціалізованих ГАК PCU з певною кількістю датчиків та виконавчих механізмів. Окрім очевидних переваг наявності одного універсального ГАК PCU, існує недолік у вигляді наявності невикористаних деталей, коли ГАК використовується або використовується лише з одним датчиком та виконавчим механізмом, що призводить до зайвих накладних витрат. З іншого боку, спеціалізовані ГАК PCU потребують більше ресурсів на проектування та обслуговування, інтерфейсу компонента GAC PCU показано на рисунку 3.8.

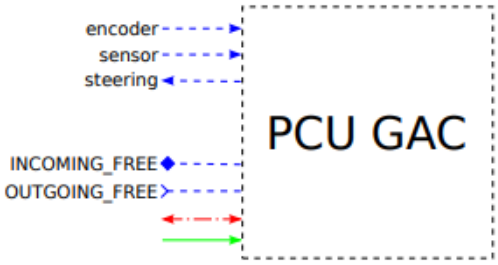

Рисунок 3.8 - Інтерфейс GAC для виробничого блоку

Зауважте, що це лише один з кількох варіантів інтерфейсу. Він має періодичні вхідні сигнали для енкодера та датчиків матеріалу, а також періодичні вихідні сигнали для керуючих значень. GAC має визначені користувачем вхідні та вихідні дані на основі подій. Подія OUTGOING_FREE використовується для

отримання інформації про те, чи приймає наступний сусід шматок матеріалу. Подія INCOMING_FREE генерується компонентом, якщо він приймає вхідний шматок матеріалу від попереднього сусіда.

Кожен алгоритм керування PCU може бути реалізований за допомогою PID-регулятора, який відповідає отриманому ним заданому значенню. Генератор профілю руху періодично надає задане значення з одного з доступних профілів руху. Активний профіль руху залежить від завдання PCU, наприклад, може знадобитися запустити виконавчий механізм, підтримувати його роботу або зупинити. Використання компонента PID та профілю руху таке ж, як і в прикладах.

Фокус фактичної реалізації проектування ГАК PCU може варіюватися між *точкою зору моделювання* або *точкою зору виконання.* Фокус на чистому моделюванні призводить до детальної моделі ГАК PCU, що складається з ГАК профілю руху та ГАК PID для реалізації обчислень. Фокус на точці зору виконання призводить до мінімалістичного використання ГАК, а реалізація обчислень складається з оптимізованого вихідного коду.

Точка зору моделювання

Реалізація PCU GAC, зосереджена на моделюванні, показана на рис. 3.9. Вона має два підпрофілі GA C, подібні до тих, що на рисунку 3.4. Хоча в цьому випадку профіль руху GAC має набір попередньо визначених профілів руху, один з яких вибирається залежно від необхідного завдання PCU. Конфігурація гачка детектора помилок не включені до рисунка для деякої чіткості, але вони використовуються для своїх звичайних завдань.

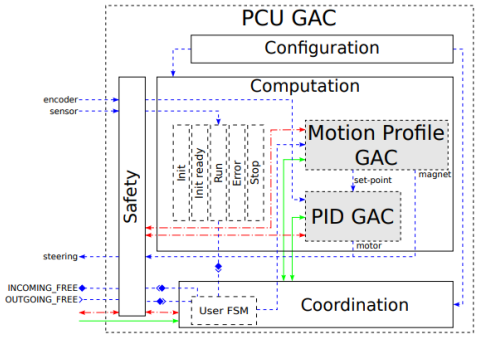

Рисунок 3.9 - Реалізація PCU GAC з точки зору моделювання

Визначений користувачем сигнал на основі події додається між блоком обчислення та блоком координації, щоб повідомити блок координації, коли шматок матеріалу проходить повз датчик. Подія генерується в перехоплювачі Run, який використовує зміни вхідних сигналів датчика для перевірки, чи шматок матеріалу проходить повз датчик. Подія, що належить датчику, який знаходиться поблизу кінця шляху матеріалу PCU, використовується для повідомлення визначеного користувачем кінцевого автомата про те, що шматок матеріалу майже залишає PCU. Якщо це так, і наступний сусід приймає новий шматок матеріалу, нічого не змінюється. В іншому випадку визначений користувачем кінцевий автомат припиняє переміщення матеріалу, доки наступний PCU знову не прийматиме нові матеріали, щоб запобігти зіткненням матеріалів. Подія датчика, що представляє ситуацію на початку шляху матеріалу, використовується визначеним користувачем кінцевим автоматом для генерації події INCOMING_FREE.

Генератор профілів руху GAC періодично надає свої задані значення, використовуючи один із доступних профілів руху, залежно від активного завдання PCU GAC. Користувацький кінцевий автомат надає детальну інформацію про активне завдання, яке визначається за допомогою події OUTGOING_FREE та позицій матеріалу в GAC, як описано вище. Якщо GAC також керує електромагнітом, вибраний профіль руху також надає сигнал для ввімкнення магніту перед рухом руки, щоб фактично підняти шматок матеріалу.

ПІД-регулятор GAC використовує сигнали заданого значення та енкодера для розрахунку сигналу керування двигуном для керування виконавчим механізмом (двигуном) PCU. Сигнал електромагніту не подається на ПІД-регулятор GAC, а безпосередньо використовується як вихідний сигнал.

Якість проектування реалізації PCU GAC є високою з точки зору моделювання: усі проблеми чітко розділені, можливе повторне використання під-GAC, а сам GAC змодельовано чітко.

З іншого боку, з точки зору виконання, ця реалізація має деякі недоліки. Наприклад: Обидва під-GAC мають чітко розділені всі свої завдання, оскільки вони мають ті самі реалізації, що й на рисунку 4.5. Тому сигнали керування

перевіряються двічі: один раз у блоці безпеки під-GAC та один раз у блоці безпеки GAC PCU. Профіль руху просто повинен надати задані значення для вибраного профілю, тому дуже малоймовірно, що це призведе до будь-яких помилок, тому сигнал заданого значення навіть не потребує перевірки взагалі. GAC профілю руху також не потребує ініціалізації, тому його кінцевий автомат життєвого циклу містить деякі невикористані стани, такі як стани Init та Error .

Точка зору виконання

Реалізація PCU GAC з точки зору виконання показана на рисунку 3.10. Основна відмінність між обома реалізаціями GAC полягає в тому, що GAC з точки зору виконання більше не має під-GAC. Їхня функціональність об'єднана з перехоплювальними функціями обчислювального блоку, що призводить до втрати розділення завдань та можливості повторного використання. Наприклад, функціональність профілю руху та PID потребує переробки, якщо інший GAC потребує такої функціональності, оскільки їхні окремі реалізації об'єднуються та більше не можуть використовуватися окремо. Вони також більше не оновлюються, для цього простого випадку це не є великою проблемою, але коли GAC стають складнішими, бажано підтримувати їх в актуальному стані з усіма виправленнями та покращеннями. Іншим недоліком є те, що розділення між жорстким та м'яким реальним часом більше неможливе, оскільки об'єднані реалізації взаємопов'язані.

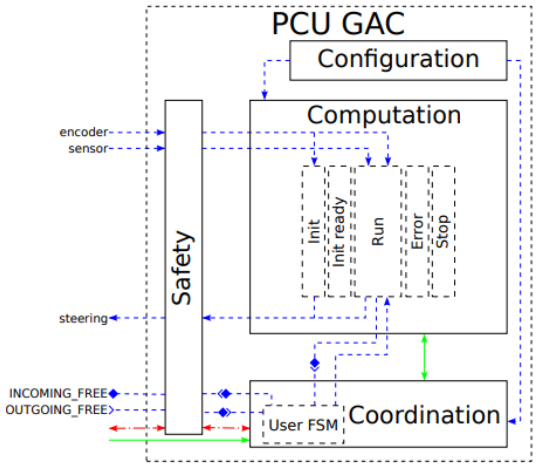

Рисунок 3.10 - Реалізація PCU GAC з точки зору виконання

З точки зору виконання, ця реалізація справді є ефективнішою. Існує лише один блок безпеки, використовуються всі стани життєвого циклу та потрібно менше внутрішнього зв'язку, оскільки, наприклад, згенеровані задані значення безпосередньо доступні PID-частині реалізації, оскільки і профіль руху, і PID-реалізація розміщуються в гачку Run PCU GAC.

Керівні принципи дизайну

Керівні принципи проектування зосереджені головним чином на визначенні рівня деталізації моделей. Це залежить як від завдань, для яких розроблена модель, так і від вимог програмного забезпечення, що працюють у режимі реального часу.

Рівень деталізації реалізації знаходиться десь посередині між точками моделювання та виконання, залежно від спеціалізованого GAC, який потрібно створити. Наприклад, відносно простий PCU GAC може бути реалізований скоріше з точки зору виконання, ніж з точки зору моделювання. Однак, у ситуації, коли оптимізоване виконання компонента не відіграє ролі, а доступний PID та/або GAC профілю руху, це економить зусилля проектування, пов'язані з повторним використанням GAC для реалізації PCU GAC.

При виборі рівня деталізації слід пам’ятати про це:

- Для ілюстрації або освітніх цілей слід використовувати точку зору моделювання.
- Для високооптимізованого програмного забезпечення, наприклад, розробленого для вбудованих обчислювальних платформ, слід використовувати точку зору виконання.
- В іншому випадку це залежить від наявності часткового рішення для реалізації компонента, його складності та досвіду проектувальника.

Зрештою, чим ближчий рівень деталізації до точки зору моделювання, тим краще, оскільки це, ймовірно, запобігає проблемам на пізніших етапах.

Зверніть увагу, що рівень деталізації можна змінити, наприклад, оптимізувавши проєкт шляхом подальшого об'єднання компонентів. Однак рекомендується робити це лише за допомогою (автоматизованих) інструментів оптимізації

та лише якщо це необхідно для запобігання проблемам використання ресурсів на (вбудованих) обчислювальних платформах.

Рівень деталізації також залежить від вимог щодо гарантій реального часу компонента. З точки зору виконання, реалізація PCU GAC призвела до повної реалізації жорсткого реального часу. Через об'єднання більшої частини його функціональності в перехоплювачі (hooks), більше неможливо використовувати різні гарантії реального часу, оскільки реалізація перехоплювача на C++ дозволяє лише один тип гарантій реального часу. Наприклад, якщо компонент вимагає жорстких гарантій реального часу для свого закону керування та м'яких гарантій реального часу для адміністративних цілей, об'єднати ці дві дії неможливо.

Впровадження архітектури виробничої комірки

реалізації програмного забезпечення керування, здатного керувати виробничою коміркою, потрібні 3 додаткові компоненти :

- Диспетчерський контролер для керування життєвим циклом 6 ГАК PCU.
- Інтерфейс користувача для відображення стану виробничої комірки користувачеві та відображення сповіщень, коли це можливо.
- Апаратний інтерфейс для доступу до датчиків та виконавчих механізмів з програмного забезпечення.

Архітектурна модель програмного забезпечення керування, що містить усі GAC та їх склад, показана на рисунку 3.11.

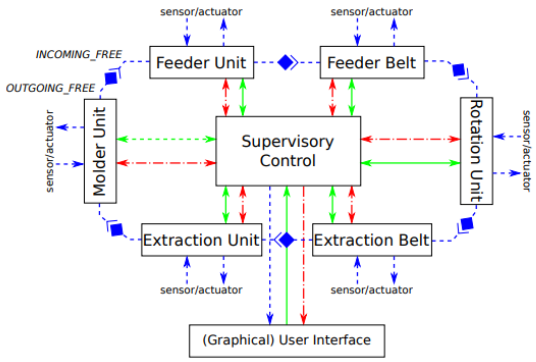

Рисунок 3.11 - Реалізація архітектури Production Cell з використанням GAC

Апаратний інтерфейс явно не показаний для зрозумілості рисунка, але він складається з одного великого блоку обробки вводу/виводу, до якого підключені всі канали датчиків та виконавчих механізмів, подібно до того, що використовується на рисунку 3.4.

На рисунку показано кільцеподібне з'єднання між 6 контролерами глобальної атаки (GAC) PCU, що складається з підключених каналів подій INCOMING_FREE та OUTGOING_FREE контролерів глобальної атаки PCU. Крім того, кожен GAC PCU підключений до власного набору виконавчих механізмів та датчиків за допомогою апаратного інтерфейсу, який не враховано.

Система керування підключена до PCU для надсилання команд для синхронізації їхніх життєвих циклів. Сигнали помилок використовуються для повідомлення системи керування про глобальні помилки, щоб вона могла коректно завершити роботу системи, коли виникла невиправна проблема.

Інтерфейс користувача підключено до системи керування. Він має командне з'єднання для надсилання команд до контролера інформаційних даних (GAC) системи керування, щоб впливати на життєвий цикл системи, наприклад, для її запуску. З'єднання користувача та даних використовується для надання інформації про стан системи. У цій конструкції немає з'єднань користувача та даних від GAC PCU до GAC системи керування, тому дані, що надсилаються до інтерфейсу користувача, не можуть містити інформацію про виконавчі механізми або датчики. Якщо це необхідно, між GAC PCU та GAC системи керування необхідно включити деякі додаткові сигнали .

Сигнал помилки використовується для сповіщення інтерфейсу користувача про невиправні помилки, щоб оператор міг обробити їх вручну. Обробка помилок у програмному забезпеченні керування здійснюється локально та глобально, як це пропонується шаблоном проекту GAC.

Прикладом локальної проблеми є ситуація, коли матеріал застряг між дверцятами формувальної машини та стінкою: блок безпеки виявляє це, оскільки супровідний кодер не показує очікуваний рух дверцят. Блок безпеки формувального блоку переводить GAC у стан помилки, надсилаючи подію ERROR, тому

він належним чином вимикається, щоб запобігти (подальшому) пошкодженню. Це приклад локальної проблеми, оскільки решта системи може продовжувати функціонувати належним чином, оскільки, наприклад, все ще має сенс підтримувати роботу екстракційного конвеєра для транспортування оброблених шматків матеріалу до місця зберігання або, в цій змодельованій системі, до оберального блоку.

Через цю помилку блок подачі також автоматично зупиниться, оскільки сигнал OUTO-ING_FREE формувального блоку GAC ніколи не вказує на прийняття нового шматка матеріалу. Таким чином, на стрічковому подачі виникає затримка, і виробнича комірка зрештою повністю зупиниться безпечним способом. З цього прикладу зрозуміло, що системі диспетчерського керування GAC не потрібно вимикати інші GAC блоку PCU, але в інших ситуаціях це може знадобитися.

Прикладом глобальної проблеми є ситуація, коли витяжний важіль зачіпається в зоні формування, блокуючи відкриті дверцята формування. Формувальний блок не може виявити цю проблему, оскільки не має для цього датчика, але він повинен залишати дверцята відкритими, щоб запобігти пошкодженню дверцят та витяжного важеля. Витяжний важіль здатний виявити проблему, порівнюючи значення свого енкодера з очікуваними значеннями. Коли ці значення не збігаються, необхідно повідомити систему диспетчерського керування, щоб вона могла вимкнути формувальний блок. Решта системи все ще може функціонувати належним чином, хоча затримка також виникатиме в цій ситуації.

Усі вимоги GAC виконано:

- Він надає кілька методів, таких як перехоплювачі та підтримка конфігурації, щоб забезпечити засоби повторного використання як шаблонного GAC, так і його спеціалізованих GAC, Вимога 1.
- Він використовує узагальнені порти для потреб зв'язку, як це передбачено методологією CPC, Вимога 2.
- Він підтримує кілька гарантій у режимі реального часу одночасно та в межах одного компонента, Вимога 3.

- Він має внутрішню підтримку безпеки та механізми для вирішення ситуацій, пов'язаних з безпекою, як на місцевому, так і на глобальному рівні, Вимога 4.
- Використовуючи надані перехоплюючі елементи, можна розробити багато різних типів спеціалізованих глобальних акцій (GAC) на основі шаблону GAC, Вимога 5.
- Формальна верифікація частково можлива, оскільки GAC розроблено з використанням моделей CSP, Вимога 6. На жаль, життєвий цикл FSM недоступний у CSP, тому повна формальна верифікація поки що неможлива.
- Може використовуватися в ієрархічних мережах, Вимога 7.

Порівняно з попередньою реалізацією універсального компонента, поточна конструкція GAC є більш гнучкою та багаторазовою, як це диктується Вимогою 1. Вона може бути використана для широкого спектру різних типів кіберфізичних систем. Спеціалізовані GAC можуть бути використані багаторазово в межах однієї системи або між різними системами завдяки можливостям конфігурації.

Порівняно з компонентом Orocos, GAC дійсно надає менше функцій і, -ймовірно, тому його менше можна використовувати повторно, що є логічним результатом. Цьому зменшенню можливості повторного використання протидіє забезпечення:

- можливості додавання функціональності, визначеної за використанням, у кількох місцях у GAC
- підтримка різних гарантій у режимі реального часу в рамках GAC
- блок власної безпеки

Ці функції збільшують (повторну) можливість використання, водночас зменшуючи невикористану функціональність, оскільки вони, ймовірно, будуть використовуватися у всіх спеціалізованих глобальних обчислювальних центрах (ГАК). І останнє, але не менш важливе: низьке споживання ресурсів завдяки менш складній конструкції ГАК робить його більш придатним для використання на вбудованих обчислювальних платформах.

Початкові впровадження та візуальні перевірки показують, що GAC працює належним чином. Виробнича комірка функціонує належним чином і належним чином транспортує матеріали через систему. Отже, можна зробити висновок, що шаблон GAC підходить для використання для спеціалізованих GAC, і ці спеціалізовані GAC можна використовувати повторно для реалізації різних частин системи, лише налаштовуючи компоненти таким чином, щоб вони поводилися належним чином для їхньої конкретної частини системи. Таким чином, можна зробити висновок, що GAC дійсно є універсальним компонентом, який можна використовувати для визначення архітектури системи та забезпечення реалізації програмного забезпечення керування кіберфізичною системою.

У підрозділі наведено правила, що допомагають визначити рівень деталізації реалізації. З одного боку, це з точки зору моделювання, а з іншого – з точки зору виконання. Також обговорюється, що оптимізація проекту не рекомендується, якщо це явно не вимагається через брак достатніх ресурсів обчислювальної платформи. Якщо оптимізація моделі все ще потрібна, рекомендується використовувати інструментальну підтримку для цього завдання.

Ефективне використання GAC вимагає використання інструменту проектування на основі моделі, який допоможе інженеру використовувати шаблон GAC під час проектування спеціалізованого GAC. Наразі шаблон GAC має свої перехоплюючі елементи, які можна використовувати для додавання реалізацій, визначених користувачем, для певних аспектів спеціалізованого GAC. Ці аспекти, визначені користувачем, потребують додаткових даних для обробки, інакше вони не зможуть зробити свій внесок у загальну систему. Додавання цих аспектів, визначених користувачем, вручну суперечить меті GAC та способу його роботи. Інструмент MDD повинен автоматизувати ці ручні завдання для оптимізації процесу проектування.

Такий інструмент MDD описано але він поки що не має інтегрованої підтримки для GAC. Підтримка GAC повинна забезпечувати засоби для додавання реалізацій до перехоплювачів GAC, які використовуються під час генерації коду до вихідного коду LUNA. Таку підтримку необхідно включити, щоб мати змогу

фактично спробувати спосіб роботи в поєднанні з GAC. Таким чином, можна перевірити, чи справді їхня комбінація добре працює, чи потрібно внести корективи до будь-якого з них.

Архітектура LUNA

LUNA розроблена модульним способом, вона має компоненти, які можна відключити, якщо вони не потрібні. Крім того, компонент може мати залежності від інших компонентів, а це означає, що для реалізації компонента потрібна реалізація інших компонентів. На рисунку 3.12 показано огляд LUNA.

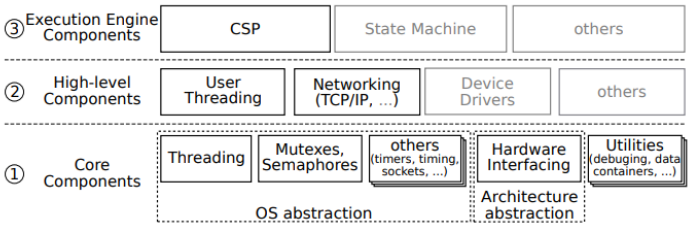

Рисунок 3.12 - Огляд архітектури LUNA.

Компоненти, згруповані за різними рівнями. Сірі компоненти ще не реалізовані, але планується їх впровадження в майбутніх випусках.

Основні *компоненти* Рівень ф містить базові компоненти, що здебільшого складаються з компонентів підтримки платформи, що забезпечують загальний інтерфейс для специфічних для платформи функцій. *Компоненти абстракції ОС* надають інтерфейси та реалізацію для специфічних для операційної системи функцій, таких як потоки, м'ютекси, таймери та синхронізація. На момент написання доступна підтримка QNX, Linux та частково Xenomai. *Компоненти абстракції архітектури* забезпечують підтримку функцій, специфічних для архітектури (або апаратної платформи), таких як підтримка можливостей (цифрового) введення та виведення (I/O). Інші компоненти повинні використовувати ці основні компоненти для використання специфічних для платформи функцій без знання фактично обраної платформи або операційної системи. Інша група основних компонентів не є компонентами абстракції платформи чи операційної системи, але надає функції низького рівня, такі як налагодження, загальні

інтерфейси та контейнери даних. Вони називаються *утилітними компонентами.*

Наступний рівень містить *високорівневі компоненти* <2. Ці компоненти не залежать від платформи, оскільки вони використовують функціональність основних компонентів. Наприклад, мережевий компонент, що забезпечує мережеві функції та протоколи, використовує сокетний компонент як платформозалежний зв'язок та надає (високорівневі) компоненти протоколу.

Компоненти *механізму виконання* @ забезпечують підтримку механізму виконання. Ці компоненти використовуються для визначення потоку виконання програми. Наприклад, компонент механізму виконання CSP надає конструкції для потоку виконання на основі CSP. Компонент CSP використовує основні компоненти для підтримки потоків або м'ютексів. Він використовує високорівневі компоненти, такі як потоки користувача, для зменшення кількості потоків ОС для покращення швидкості виконання , а мережевий компонент - для реалізації каналів зустрічі через мережу.

Компоненти можна вмикати або вимикати у фреймворку, залежно від типу програми, яку потрібно розробити. Невикористовуваний компонент можна вимкнути, щоб заощадити ресурси та використовувати їх в інших аспектах програми, як того вимагає Вимога 11. Оскільки збірка LUNA є складною через компонентний підхід та різноманітність підтримуваних платформ, передбачено спеціальну систему збірки. Система збірки LUNA значною мірою базується на системі збірки OpenWrt.

OpenWrt надає користувацьку прошивку, що складається з ядра Linux, інструментів операційної системи та програм, зібраних з нуля для всіх типів маршрутизаторів. Інструменти та програми можуть бути ввімкнені користувачами залежно від їхніх потреб. Підтримувані маршрутизатори мають різноманітне обладнання, що вимагає відповідної підтримки драйверів. Для зборки прошивки потрібні методи крос-компіляції, оскільки неможливо зібрати її на самих маршрутизаторах. Усе це разом робить створення прошивки складним завданням, яке виконує система збірки OpenWrt. Згадані особливості OpenWrt відповідають потребам LUNA; він також вимагає підтримки різного обладнання, модульності

компонентів та кінцевого результату, подібного до прошивки.

Першою операційною системою, яку підтримує LUNA, є QNX. Це мікроядерна ОС реального часу, яка нативно підтримує жорсткий реальний час та зв'язок між операційними системами. Її базовий набір функцій зняв навантаження з розробки для початкового впровадження LUNA, оскільки вона охоплювала кілька вимог.

Крім того, QNX сумісний з *інтерфейсом портативних операційних систем* (POSIX). Це набір стандартів для підтримки сумісності між операційними системами. Таким чином, реалізація QNX для LUNA забезпечує підтримку POSIX для LUNA. Багато інших операційних систем також повністю або переважно сумісні з POSIX, тому додавання підтримки для цих операційних систем не вимагає особливих зусиль. Linux є такою операційною системою, вона повторно використовує більшу частину реалізації POSIX платформи QNX.

На пізнішому етапі створення реалізації LUNA механізм зв'язку операційної системи QNX rendezvous замінюється реалізацією власного коду, оскільки неможливо використовувати канали QNX між двома потоками користувача, що працюють в одному потоці ОС.

Використання TERRA

TERRA було розроблено для сприяння процесу розробки програмного забезпечення для керування кіберфізичними системами, як описано в способі його роботи. У цьому розділі показано, що TERRA дійсно можна використовувати для цих цілей. Магістранти успішно використовують TERRA вже два роки для розробки програмного забезпечення для керування установкою JIWY для курсу «Розробка програмного забезпечення в реальному часі». Це показує, що TERRA підходить для початківців-користувачів для розробки відносно невеликих моделей для керування простими кіберфізичними системами. У перший рік роботи TERRA виникло кілька «проблем», починаючи від проблем сумісності платформи (Windows) і закінчуючи відсутніми функціями, що заважали студентам розробляти свої моделі так, як вони хотіли. На другий рік використання TERRA він був значно покращений, і жодних серйозних проблем не повідомлялося.

TERRA також використовувався для розробки першої реалізації GAC на базі CSP, більше деталей про цю реалізацію наведено в підрозділі.

У TERRA виникло кілька «проблем», цього разу головним чином через проблеми з обробкою великих моделей, такі як копіювання/вставка великих частин проекту для повторного використання в іншому місці або в новій версії моделі. Ці проблеми також були вирішені, і початкова реалізація GAC показує багатообіцяючі результати. Таким чином, можна зробити висновок, що TERRA підходить як набір інструментів розробки для дослідження структур та реалізацій моделей.

Початкова реалізація GAC CSP була використана для фактичного керування значною частиною виробничої комірки. Це показує, що TERRA також підходить для більших моделей, порівняно з установкою JIWY, для керування кіберфізичною системою в дослідницьких цілях.

Велика кількість процесорного часу, необхідного для виконання реалізації GAC, виявилася проблемою для програмної платформи, на якій вона мала виконуватися, звідси й часткова реалізація. Основна причина полягає в тому, що реалізація GAC була перевіркою принципу, без будь-яких оптимізацій, щоб показати, що ідеї, що лежать в основі GAC, насправді є розумними. Тим не менш, це показує, що TERRA є потужним набором інструментів моделювання для розробки програмного забезпечення керування кіберфізичними системами.

Обговорення

Генерація коду виявилася корисною, особливо для захищених областей, оскільки TERRA поки що не надає редактора моделей машин станів для реалізації станів життєвого циклу або підтримки взаємодії з апаратним забезпеченням Production Cell з моделей. Те саме стосується інструментів валідації моделей, оскільки такі великі моделі неможливо розробити за один раз, тому деякі області залишаються порожніми, і ці інструменти валідації допомагають знайти частини моделей, які ще потрібно заповнити.

Впровадження Виробничої комірки також показало, що деякі інструменти явно бракували. Не було доступного механізму моделювання для тестування

(проміжних) моделей GAC та Виробничої комірки, як це передбачалося способом роботи. Як обхідний шлях, згенерований код був протестований на віртуальній платформі, але це виявилося досить складним, і отриманій якості бракувало деталей, щоб довести, що програмне забезпечення керування повинно працювати на реальній кіберфізичній системі, не порушуючи її.

Деякі проблеми, такі як глухі блокування, виникали під час тестування програмного забезпечення керування. Через розмір і складність моделей реалізації, наприклад, було майже неможливо з'ясувати, які частини постраждали від глухих блокувань і що їх спричинило. Інтеграція інструментів підтримки журналювання до TERRA допомагає вирішити ці проблеми, оскільки отриману інформацію можна використовувати для анімації моделей,. Анімація моделі забезпечує підсвічування синтаксису та анотації для моделей, щоб відобразити поточний стан запущеного програмного забезпечення. Коли програмне забезпечення призупинено через взаємодію з користувачем або проблеми, такі як глухі блокування, ця інформація допомагає показати проблемні області моделі.

Крім того, в TERRA відсутня підтримка GAC. Це головним чином пов'язано з незрілістю реалізації GAC та її схильністю до змін. Але ця відсутність підтримки значно ускладнила реалізацію Production Cell, оскільки реалізації GAC потрібно було копіювати/вставляти з головного шаблону та вручну модифікувати відповідно до ситуації , для якої вони використовувалися. Прикладом ручних модифікацій є зміна кількості користувацьких портів та їх підключення до відповідних частин GAC. Ці ручні зміни є виснажливими та займають багато часу у розробника. Підтримка GAC зможе впоратися з необхідними змінами залежно від параметрів конфігурації.

3.3 Аналіз результатів функціонування та оцінка ефективності системи

Проектування програмного забезпечення для керування сучасними кіберфізичними системами має тенденцію до ускладнення через зростання

складності цих систем. Головною метою цієї роботи є надання рекомендацій щодо управління складністю програмного забезпечення. Ці запропоновані рекомендації забезпечують структурний підхід до роботи, який базується на методах проектування на основі моделей. Мета цього підходу - забезпечити засоби для розробки правильної реалізації програмного забезпечення для керування з першого разу. Моделі, що лежать в основі проектування програмного забезпечення для керування, допомагають зрозуміти та осягнути зростаючу складність кіберфізичних систем та їх конструкцій.

Спосіб роботи базується на розділенні обов'язків. На системному рівні це забезпечується шляхом використання окремих компонентів для кожної (керуючої) частини програмного забезпечення. Компонент загальної архітектури надає план або шаблон для цих програмних компонентів. Розділення обов'язків також застосовується в розробці шаблону GAC, щоб зробити його зрозумілим та зручним для використання спеціалізованими GAC.

Фреймворк виконання LUNA розроблений для виконання змодельованого керуючого програмного забезпечення. Виконавчі механізми складаються зі статичних частин для підтримки визначень метамоделей, таких як конструкції CSP, планувальник та апаратна підтримка. Набір інструментів під назвою TERRA надає графічні способи створення, маніпулювання та перевірки моделей програмного забезпечення. Зрештою, для перетворення моделей у програмне забезпечення використовуються перетворення моделі в код. Програмне забезпечення використовує фреймворк виконання, тим самим зменшуючи складність перетвореного коду.

Цей спосіб роботи передбачає кроки для повної траєкторії проектування програмного забезпечення. Його проблеми розділені відповідно до підходу Формалізм, Методи, Методи та Інструменти, щоб зробити його максимально універсальним. Наприклад, він використовує метамоделі для

визначити семантику моделей, а не дозволяти інструментам обробляти її самостійно. Крім того, реалізація інструментів не залежить від фактичних кроків, тобто будь-який набір інструментів може бути використаний з урахуванням

способу роботи, якщо їхня функціональність є достатньою.

Етапи проектування моделі в рамках цього методу роботи забезпечують спосіб побудови моделей з використанням моделей динамічних об'єктів та законів керування для реалізації компонентів (контурного керування). Метод роботи також включає кроки перевірки та моделювання розроблених програмних моделей, тим самим підвищуючи якість результуючого програмного забезпечення.

Спосіб роботи починається з проектування архітектури програмного забезпечення керування. Фактична реалізація архітектурних компонентів доповнюється на наступних кроках. Такий підхід гарантує, що архітектура системи відповідає фізичній системі та її окремим завданням. Можна використовувати різні реалізації компонентів, якщо їхні інтерфейси відповідають інтерфейсу компонента архітектурної моделі, наприклад, для дослідження простору проектування, моделювання або налагодження.

Цей спосіб роботи підходить для широкого спектру кіберфізичних систем, від складних медичних систем до простих дослідницьких установок. Масштабованість цього способу роботи забезпечується тим, що кроки частково необов'язкові, тобто не потрібно виконувати кожен крок точно. Певний крок можна повністю пропустити або кілька кроків можна об'єднати та виконати одночасно, коли це дозволяє складність проекту.

Траєкторія проектування програмного забезпечення поєднує та повторно використовує інформацію з траєкторій проектування електроніки, механіки та контролера на кількох етапах роботи. Це ще більше підвищує якість результуючого програмного забезпечення для керування. Тим самим також збільшуються шанси на успішне проектування з першого разу, оскільки програмне забезпечення для керування краще інтегроване та узгоджується з іншими частинами кіберфізичної системи.

Розроблений шаблон GAC тісно відповідає способу роботи, що підвищує його цінність та зручність використання. Завдяки своїй компонентній природі, реалізація GAC взаємозамінна з іншими реалізаціями, якщо їхні інтерфейси збігаються, як описано вище. Хоча у випадку GAC їхня функціональність також

повинна збігатися. Наприклад, якщо GAC очікує, що певні командні команди, визначені користувачем, функціонуватимуть належним чином, GAC на заміну повинен очікувати тих самих команд, інакше програмне забезпечення не працюватиме належним чином.

Масштабованість та можливість повторного використання GAC досягаються завдяки цим трьом основним причинам:

- Шаблонний GAC не підтримує всі види розширеного функціоналу, як компонент Orocos. Це робить його придатним для вбудованих програм, що потребують низького використання ресурсів. З іншого боку, розширений функціонал легко включається за допомогою наданих перехоплювачів, що робить спеціалізований GAC придатним для більш складних програм, якщо це необхідно. Таким чином, повним спектром кіберфізичних систем можна керувати за допомогою компонента загальної архітектури.
- Різноманітність наданих перехоплювальних функцій дозволяє розробнику додавати будь-яку необхідну реалізацію до спеціалізованого GAC без зміни основ GAC. Це робить GAC придатним для будь-якого типу застосувань керування, починаючи від контролерів циклів реального часу і закінчуючи складними завданнями, такими як картографування середовища, пошук шляхів або планування завдань.

• Блок конфігурації підвищує можливість повторного використання спеціалізованих глобальних контролерів даних (GAC), оскільки (невеликі) відмінності між двома компонентами контролера можна застосовувати за допомогою конфігурації компонентів без необхідності проектування окремих компонентів.

Підхід розділення проблем, що застосовується до проектування GAC, допомагає проектувальнику використовувати правильну проблему для правильного завдання. Наприклад, блок координації містить всю логіку прийняття рішень у вигляді двох кінцевих автоматів: попередньо визначеного кінцевого автомата з циклом виконання та користувацького кінцевого автомата для додавання більш детальних використовуваних станів до спеціалізованого компонента. Підтримка перевірки вхідних та вихідних сигналів забезпечується в блоці

безпеки, який має перехоплювач, щоб користувач міг легко додавати необхідні перевірки. Це підвищує ймовірність запобігання непередбаченим ситуаціям. Як згадувалося вище, блок конфігурації підвищує можливість повторного використання компонентів. Додатковою перевагою є те, що він також допомагає запобігти помилкам проектування, оскільки при повторному використанні компонентів вони, ймовірно, містять менше помилок, ніж щойно розроблені компоненти.

Усі ці окремі переваги використання GAC як основи для компонента збільшують шанси на успіх підходу, що робить все правильно з першого разу.

Платформа виконання LUNA та набір інструментів TERRA забезпечують підтримку для легкого проектування, перевірки та тестування моделей програмного забезпечення.

Редактор архітектурних моделей інструментального пакету TERRA забезпечує підтримку дослідження простору проектування. Його результуючі моделі зазвичай описують системну архітектуру кіберфізичної системи та її програмного забезпечення, повністю відокремлено від фактичної реалізації. Залежно від вимог, наприклад, чи використовується модель для виробничої реалізації, чи для тестування чи моделювання, може бути обрана фактична реалізація.

Компонентно-орієнтований підхід LUNA використовується для вибору підтримки фреймворку, необхідної для застосунку. Рівень абстракції апаратного забезпечення LUNA використовується для забезпечення специфічної підтримки використовуваного обладнання та операційної системи різних обчислювальних платформ. Все це підвищує масштабованість фреймворку, роблячи його придатним для широкого спектру систем.

Метамоделі TERRA значно допомогли в розробці TERRA з різних причин. Використання метамоделей змушує розробника задуматися про семантику моделі перед фактичним впровадженням набору інструментів TERRA. Крім того, завдяки чітким визначенням моделі зрозуміло, де і як інструмент повинен отримати необхідну інформацію. Автоматична валідація моделі, що забезпечується EMF, використовує семантику метамоделі для визначення правил валідації, тому розробник моделі опосередковано отримує допомогу від метамоделей,

а самі метамоделі допомагають запобігти помилкам програмного забезпечення.

Цей спосіб роботи передбачає кроки проектування, які можуть бути використані іншими для належного проектування програмного забезпечення керування кіберфізичною системою. Практична реалізація інструменту забезпечується поєднанням GAC, LUNA та TERRA.

Цей спосіб роботи та супутня підтримка мають на меті скоротити час розробки програмного забезпечення для керування та підвищити якість результуючого програмного забезпечення. Компанії можуть використовувати цю роботу для покращення своїх вимог щодо часу виходу на ринок та випередження конкурентів.

Академічна спільнота може впроваджувати точку зору моделювання та її рекомендації, щоб навчити студентів структурованій роботі. Це допомагає їм зрозуміти та вивчити концепції моделювання та його переваги, не намагаючись вирішити всілякі проблеми через нечіткі моделі.

Навіть попри висновок, що цей спосіб роботи в поєднанні з GAC, LUNA та TERRA допомагає керувати складністю програмного забезпечення керування, покращуючи можливість повторного використання та забезпечуючи численні інші переваги, все ще існують можливості для подальшого покращення їхньої якості. Тут обговорюються рекомендації, що мають найбільший вплив.

Спосіб роботи повністю охоплює всю траєкторію проектування та дозволяє використовувати різні підходи залежно від вимог до проектування кіберфізичної системи. У розділі описано кілька різних типів кіберфізичних систем, а саме: медичні, промислові, малі вбудовані та дослідницькі кіберфізичні системи. Програмне забезпечення керування для всіх цих систем може бути розроблене з урахуванням цього способу роботи. Тим не менш, це ще не було чітко протестовано для всіх цих типів кіберфізичних систем у різних ситуаціях.

Після впровадження інших варіантів використання та оцінки процесу проектування, ймовірно, будуть знайдені моменти для подальшого покращення способу роботи. Коли отримано більш зрілу версію способу роботи, його кроки проектування також необхідно уточнити, щоб отримати стандартизовану

структуру, яка краще підходить для загального використання.

Таке подальше вдосконалення методу роботи призводить до вищої якості програмного забезпечення, що ще більше наближає нас до ідеалів «правильного виконання з першого разу». Крім того, це також призводить до швидшого та ефективніший дизайн програмного забезпечення для керування, що скорочує час виведення нових продуктів на ринок.

Одна з причин використання компонентних інтерфейсів в архітектурних моделях кіберфізичної системи полягає в можливості змінювати фактичну реалізацію залежно від поточного використання моделі. З одним або двома компонентами це чудово підтримується, але якщо кількість компонентів більша, перемикання всіх реалізацій стає проблематичним.

Зручно відстежувати кожен набір реалізацій, щоб їх можна було перемикати як єдине ціле. Результатом проекту DESTECS стала система управління моделями, яка використовується для відстеження різних версій або реалізацій однієї й тієї ж моделі, що складається з кількох файлів . Така система управління моделями також може бути використана для відстеження набору реалізацій компонентів, що підходять для конкретної реалізації архітектурної моделі.

Розширення інструменту TERRA підтримкою системи керування моделями покращує зручність використання цього методу роботи. Повторне використання моделей для різних випадків використання та ситуацій стає легшим, що, наприклад, скорочує час, необхідний для виконання симуляцій.

Навіть попри те, що надаються рекомендації щодо проектування для вибору відповідного рівня деталізації програмного забезпечення керування, все ще важко вибрати правильний рівень. Використання з точки зору моделювання все ще розглядається як свого роду «утопія», оскільки воно має зробити модель зрозумілою з першого погляду. На практиці ці моделі непридатні для щоденного використання через складність та кількість використовуваних елементів моделі.

Методи оптимізації моделей надають спосіб змінити рівень деталізації з точки зору моделювання на точку зору виконання, або принаймні набагато ближче до цієї точки зору. Це робить моделі більш зручними для практичних

ситуацій.

Тому рекомендується включати ці методи оптимізації до інструменту перетворення моделей. Інструмент оптимізації моделей повинен викликатися автоматично, коли запускаються певні завдання, такі як генерація коду для симуляцій.

Такий інструмент міг би навіть адаптувати третю точку зору, а саме точку зору моделювання, яка визначає певну максимальну ієрархічну глибину в моделях. Це, ймовірно, покращить якість моделювання, оскільки складність зменшується, і користувачеві не доведеться турбуватися про результати моделювання базових (компонентних) реалізацій.

3.4 Висновок по розділу

У третьому розділі було розглянуто розробку та оцінку ефективності програмного забезпечення кіберфізичної системи. Проведено проєктування загальної архітектури системи з урахуванням вимог до масштабованості, продуктивності та взаємодії компонентів. Реалізовано програмне забезпечення та виконано інтеграцію основних модулів кіберфізичної системи для забезпечення коректної роботи в реальному середовищі. Також проведено аналіз результатів функціонування системи та оцінено її ефективність за основними показниками роботи. Отримані результати підтверджують доцільність використання сучасних методів проєктування та розробки програмного забезпечення для створення надійних і ефективних кіберфізичних систем.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено та розроблено програмне забезпечення кіберфізичних систем, що є комплексним процесом, спрямованим на створення надійних, масштабованих та ефективних систем керування і взаємодії між програмними та фізичними компонентами. У роботі проаналізовано сучасні підходи до побудови кіберфізичних систем, досліджено особливості роботи систем реального часу, а також реалізовано програмні компоненти, що забезпечують стабільне функціонування системи відповідно до сучасних вимог якості та безпеки.

На початковому етапі було проведено аналіз предметної області та визначено функціональні й нефункціональні вимоги до програмного забезпечення кіберфізичних систем. Особливу увагу приділено питанням надійності, продуктивності, масштабованості та забезпечення роботи в режимі реального часу. Аналіз існуючих рішень дозволив визначити ключові проблеми, пов’язані з інтеграцією програмних і апаратних компонентів, синхронізацією процесів та забезпеченням стабільної взаємодії між сенсорами, виконавчими механізмами й програмним забезпеченням.

У теоретичній частині роботи розглянуто основи кіберфізичних систем, принципи їх функціонування та особливості застосування систем реального часу. Було досліджено модельно-орієнтований підхід до розробки програмного забезпечення, який дозволяє підвищити ефективність проєктування складних систем, спростити процес тестування та забезпечити кращу адаптивність програмних компонентів до змін у системі.

У другому розділі проаналізовано методи та інструменти розробки програмного забезпечення для кіберфізичних систем. Розглянуто підходи до проєктування вбудованих систем керування, моделювання архітектури програмного забезпечення та використання засобів тестування й верифікації. Це дозволило сформувати оптимальну архітектуру системи та забезпечити ефективну взаємодію між її компонентами.

У практичній частині роботи було реалізовано архітектуру кіберфізичної системи та програмне забезпечення для взаємодії між апаратними й програмними компонентами. Було забезпечено підтримку роботи в режимі реального часу, обробку даних від сенсорів та передачу керуючих команд до виконавчих пристроїв. Особливу увагу приділено інтеграції компонентів системи, оптимізації продуктивності та забезпеченню стабільності функціонування при високих навантаженнях.

Проведено тестування програмного забезпечення, яке включало модульні, інтеграційні та функціональні тести. Результати тестування підтвердили відповідність системи поставленим вимогам та продемонстрували ефективність обраних методів і технологій. У процесі тестування було виявлено окремі аспекти, що потребують подальшої оптимізації, зокрема покращення швидкодії при великій кількості одночасних процесів і вдосконалення механізмів відмовостійкості.

Загалом, розроблене програмне забезпечення відповідає поставленим цілям і забезпечує ефективне функціонування кіберфізичної системи. Основними перевагами реалізованого підходу є гнучкість архітектури, підтримка масштабування, можливість інтеграції з різними типами апаратних компонентів і забезпечення роботи в режимі реального часу. Водночас існують певні обмеження, зокрема складність інтеграції із застарілими системами та необхідність додаткової оптимізації при роботі з великими потоками даних.

У подальшому перспективним є розширення функціональних можливостей системи шляхом інтеграції технологій штучного інтелекту, машинного навчання та хмарних сервісів для аналізу даних і автоматизованого прийняття рішень. Також доцільним є вдосконалення механізмів безпеки та відмовостійкості, що дозволить підвищити ефективність використання кіберфізичних систем у промисловості, транспорті, енергетиці та інших сферах застосування.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Lee, E. A. (2008). Cyber Physical Systems: Design Challenges. IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. https://doi.org/10.1109/ISORC.2008.25

2. Rajkumar, R., Lee, I., Sha, L., & Stankovic, J. (2010). Cyber-Physical Systems: The Next Computing Revolution. Design Automation Conference, 731–736. https://doi.org/10.1145/1837274.1837461

3. Alur, R. (2015). Principles of Cyber-Physical Systems. MIT Press. https://mitpress.mit.edu/9780262029117/principles-of-cyber-physical-systems/

4. Baheti, R., & Gill, H. (2011). Cyber-physical systems. The Impact of Control Technology, 12(1), 161–166.

5. Wolf, W. (2012). Computers as Components: Principles of Embedded Computing System Design. Elsevier. https://www.elsevier.com

6. Marwedel, P. (2021). Embedded System Design. Springer. https://link.springer.com/book/10.1007/978-3-030-60910-8

7. Burns, A., & Wellings, A. (2017). Real-Time Systems and Programming Languages. Pearson. https://www.pearson.com

8. Kopetz, H. (2011). Real-Time Systems: Design Principles for Distributed Embedded Applications. Springer. https://link.springer.com/book/10.1007/978-1-4419-8237-7

9. Sommerville, I. (2020). Software Engineering. pearson.com. https://www.pearson.com

10. Martin, R. C. (2017). Clean Architecture: A Craftsman’s Guide to Software Structure and Design. pearson.com. https://www.pearson.com/store/p/clean-architecture-a-craftmans-guide-to-software-structure-and-design/P100001465201

11. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. pearson.com. https://www.pearson.com

12. Fowler, M. (2002). Patterns of Enterprise Application Architecture. martinfowler.com. https://martinfowler.com/eaaCatalog/

13. Freeman, E., & Robson, E. (2021). Head First Design Patterns. oreilly.com. https://www.oreilly.com/library/view/head-first-design/9781492077992/

14. ISO/IEC 12207 Software Life Cycle Processes. (2024). iso.org. https://www.iso.org/standard/63712.html

15. ISO/IEC 25010 Systems and Software Quality Models. (2023). iso.org. https://www.iso.org/standard/35733.html

16. ISO/IEC 29119 Software Testing Standards. (2024). iso.org. https://www.iso.org/standard/45142.html

17. IEEE Standard for Software Verification and Validation. (2023). ieee.org. https://standards.ieee.org

18. MathWorks. (2025). Simulink Documentation. mathworks.com. https://www.mathworks.com/products/simulink.html

19. MATLAB Documentation. (2025). mathworks.com. https://www.mathworks.com/help/matlab/

20. Model-Based Design for Embedded Systems. (2024). mathworks.com. https://www.mathworks.com/solutions/model-based-design.html

21. Microsoft Azure IoT Documentation. (2025). learn.microsoft.com. https://learn.microsoft.com/en-us/azure/iot/

22. AWS IoT Core Documentation. (2025). docs.aws.amazon.com. https://docs.aws.amazon.com/iot/

23. Docker Documentation. (2025). docker.com. https://docs.docker.com/

24. Kubernetes Documentation. (2025). kubernetes.io. https://kubernetes.io/docs/

25. Jenkins Documentation. (2025). jenkins.io. https://www.jenkins.io/doc/

26. GitHub Actions Documentation. (2025). docs.github.com. https://docs.github.com/en/actions

27. Visual Studio Documentation. (2025). learn.microsoft.com. https://learn.microsoft.com/en-us/visualstudio/

28. .NET Documentation. (2025). learn.microsoft.com. https://learn.microsoft.com/en-us/dotnet/

29. C++ Reference Documentation. (2025). cppreference.com. https://en.cppreference.com/w/

30. Python Documentation. (2025). python.org. https://docs.python.org/3/

31. MQTT Protocol Specification. (2024). mqtt.org. https://mqtt.org/

32. OPC UA Documentation. (2025). opcfoundation.org. https://opcfoundation.org/about/opc-technologies/opc-ua/

33. ROS Documentation. (2025). docs.ros.org. https://docs.ros.org/

34. Stankovic, J. (2014). Research Directions for Cyber Physical Systems. ACM Transactions on Cyber-Physical Systems, 1(1), 1–7. https://doi.org/10.1145/2661569

35. Derler, P., Lee, E., & Vincentelli, A. (2012). Modeling Cyber-Physical Systems. Proceedings of the IEEE, 100(1), 13–28. https://doi.org/10.1109/JPROC.2011.2160929

36. Lee, I., & Sokolsky, O. (2010). Medical Cyber Physical Systems. 47th Design Automation Conference, 743–748. https://doi.org/10.1145/1837274.1837463

37. Hassan, M., & Khan, A. (2023). Scalable architectures for cyber-physical systems. IEEE Transactions on Industrial Informatics, 19(3), 234–248. https://doi.org/10.1109/TII.2023.3245678

38. Kumar, S., & Sharma, R. (2023). Embedded software design for industrial cyber-physical systems. Journal of Systems Architecture, 137, 102845. https://doi.org/10.1016/j.sysarc.2023.102845

39. OWASP Top Ten Security Risks. (2024). owasp.org. https://owasp.org/www-project-top-ten/

40. Selenium Documentation. (2025). selenium.dev. https://www.selenium.dev/documentation/

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: "Розробка програмного забезпечення кібер-фізичних систем"

Обсяг пояснювальної записки: 98 аркушів

Дата закінчення бакалаврської роботи 10 червня 2026р.

Підпис студента _____