Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи самовідновлення програмного забезпечення
(тема)

здобувач 2 року навчання,
групи СПм-23-5
Денис ПОЛОЗОВ
(власне ім'я, прізвище)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: проф. Максим ВОЛК
(посада, власне ім'я, прізвище)

Допускається до захисту

Зав. кафедри ЕОМ ______________ Андрій КОВАЛЕНКО
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________________
(підпис)

"____" ________________ 20 25 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Полозову Денису Максимовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Методи самовідновлення програмного забезпечення

затверджена наказом по університету від " 21 " квітня 2025 р. № 296ст

2. Термін подання студентом роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи

   1. Технології самовідновлення програмних систем
   2. Методи забезпечення надійності, функціональної стійкості та живучості програм
   3. Метод автоматичних обхідних шляхів (Automatic Workarounds)
   4. Методи самоадаптації

4. Перелік питань, що потрібно опрацювати в роботі

   1 Аналіз предметної області
   2 Методи самоадаптації та самовідновлення
   3 Експериментальні дослідження
   4 Висновки

5. Перелік графічного матеріалу із зазначенням кресленників, схем, плакатів, комп'ютерних ілюстрацій (слайдів)

Слайд-презентація – 12 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної області	22.04.25-29.04.25	
2	Розробка моделей	30.04.25-05.05.25	
3	Реалізація алгоритмів	06.05.25-10.05.25	
4	Розробка структури програмних засобів	11.05.25-21.05.25	
5	Розробка програмних модулів	22.05.25-02.06.25	
6	Оформлення матеріалів кваліфікаційної роботи	03.06.25-05.06.25	
7	Подання кваліфікаційної роботи керівникові та її попередній захист	06.06.25-10.06.25	
8	Подання кваліфікаційної роботи на рецензування	11.06.25-12.06.25	

Дата видачі завдання 21 квітня 2025 р.

Студент ____________________
(підпис)

Керівник роботи ____________________ проф. Максим ВОЛК
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 56 с., 12 рис., 1 дод., 19 джерел.

САМОВІДНОВЛЕННЯ, КРИТИЧНІ ПРОГРАМНІ СИСТЕМИ, НАДІЙНІСТЬ, ЛАНЦЮГИ МАРКОВА, ПРОГРАМНА АРХІТЕКТУРА, РОЗПОДІЛЕНІ СИСТЕМИ.

У роботі досліджується проблема забезпечення надійності критичних програмних систем шляхом впровадження механізмів самовідновлення. Запропоновано метод оцінки надійності програмного забезпечення на основі аналізу архітектури з використанням ланцюгів Маркова. Основний підхід полягає в ідентифікації ключових компонентів системи, що мають найбільший вплив на її надійність, та впровадженні для них механізмів самовідновлення. Проведені експериментальні дослідження показали, що застосування запропонованої методики дозволяє значно підвищити стійкість програмних систем до відмов і збоїв. Отримані результати можуть бути використані для розробки високонадійного програмного забезпечення, що функціонує в критично важливих галузях.

ABSTRACT

Master's thesis: 56 pages, 12 figures, 1 appendice, 19 sources.

SELF-HEALING, CRITICAL SOFTWARE SYSTEMS, RELIABILITY, MARKOV CHAINS, SOFTWARE ARCHITECTURE, DISTRIBUTED SYSTEMS.

The paper investigates the problem of ensuring the reliability of critical software systems by implementing self-healing mechanisms. A method for assessing software reliability based on architecture analysis using Markov chains is proposed. The main approach is to identify key system components that have the greatest impact on its reliability and implement self-healing mechanisms for them. Experimental studies have shown that the application of the proposed methodology allows to significantly increase the resistance of software systems to failures and malfunctions. The results obtained can be used to develop highly reliable software operating in critical industries.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧКИ.. 7

ВСТУП 8

1 Аналіз предметної області.. 11

1.1 Стійкість і живучість в комп'ютерних мережах 11

1.2 Методи оцінки надійності програмних систем.. 14

1.3 Ланцюги Маркова в оцінці надійності програмних систем 17

1.4 Життєвий цикл системи 19

1.5 Постановка мети та завдань дослідження 21

2 Методи САМОАДАПТАЦІЇ ТА САМОВІДНОВЛЕННЯ............................ 23

2.1 Модифікований життєвий цикл програмного забезпечення з самоадаптацией 23

2.2 Особливості пропонованого методу 29

3 Експериментальні ДОСЛІДЖЕННЯ.. 36

3.1 Моделювання процесів для самоадаптивних програмних систем 36

3.2 Моделювання процесу самовідновлення на основі SPEM 38

ВИСНОВКИ... 45

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 47

ДОДАТОК А 50

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧКИ

ВМ – віртуальна машина

AW – Automatic Workarounds

CDG – Component Dependency Graphs

DAG – Directed Acyclic Graph

IEEE – Institute of Electrical and Electronics Engineers

NHPP – Non-Homogeneous Poisson Process

SCADA – Supervisory Control and Data Acquisition

SPEM – Software & Systems Process Engineering Model Specification

SPN – Stochastic Petri Net

TSP – Time Stamp Protocol

UNL – Unified Modelling Language

VM – Virtual Machine

ВСТУП

Традиційні дослідження програмної інженерії в основному зосереджені на процесі розробки програмного забезпечення, а не супроводженні, обслуговування чи розвитку [1]. Однак, співтовариство розробників програмного забезпечення визнає, що програмне забезпечення повинно постійно адаптуватися та розвиватися відповідно до вимог, що постійно змінюються, щоб постійно задовольняти користувача [2]. Це усвідомлення призвело до повторюваних, еволюційних процесів у розробці програм, а не послідовному проектуванні [3], впровадженні та тестування, як це відображено у моделі водоспаду.

Однак такі підходи до розробки програмного забезпечення не відповідають вимогам сучасного контексту відносно критично важливих програмних систем або інформаційних систем великої складності [4]. Сучасні розподілені системи потребують своєчасних змін у відповідь на зміну середовища, зміни в самій системі чи її цілях. Внутрішня затримка традиційних процесів є для цих систем незадовільною. Критично важливі системи мають працювати безперервно. У традиційному процеси, зміни розгортаються під час запланованих простоїв і, як а наслідок цього, безперервна робота неможлива. Великомасштабні системи є дуже складнми, що робить діяльність, керовану людиною, складною та дорогою, або навіть нездійсненною на практиці через розмір і притаманну складність, що перешкоджає повному вимкненню системи з метою її зміни. Таким чином, можна зробити висновок, що використання традиційного процесу оновлення для таких систем несе ризик того, що система не зможе відповідати своїм специфікаціям щодо часу реакції на зміни і безперервної роботи.

Ці ризики призвели до розробки нових засобів зменшення ризиків, що змінюють програмне забезпечення в умовах самоадаптації [5]. Самоадаптивна поведінка має на увазі, що певні дії щодо розробки

програмного забезпечення переносяться на саму програмну систему, і виконуються автоматично, дублюючи діяльність інженерів програмного забезпечення або адміністраторів.

Нові принципи організації програмного забезпечення поширюються на час розробки, час розгортання та час виконання. Наслідок цієї реконцептуалізації вимагає повного переосмислення процесу розробки програмного забезпечення, де традиційний підхід заміщається новим, який об'єднує процеси розробки та виконання.

Одним з напрямком розвитку самоадаптивних систем є самовідновлення, яке полягає у властивості розподіленої програмної системи відновлювати роботу при збоях, перенавантаженнях та відмов елементів системи [6], до яких відносяться як комп'ютери, програми, так і комп'ютерні мережі.

В даній роботі розглядаються моделі та методи забезпечення самовідновлення розподілених програмних систем. Методи оцінки надійності критично важливих програмних систем, що основні на аналізі архітектури, привернули багато уваги в останні роки у зв’язку з появою моделі розробки програмного забезпечення на основі компонентів. Щоб запобігти збою програмних систем на останніх фазах розробки в критичних програмних системах, необхідно застосовувати процес оцінки надійності програмного забезпечення на всіх етапах. Оцінка надійності критичних програмних систем на основі компонентів є дуже важливою на ранніх стадіях розробки програмної системи та разом з її архітектурою стає одним з атрибутів якості програмних систем.

У даній роботі пропонується метод оцінки надійності критичних програмних систем шляхом розгляду ефекту самовідновлення компонентів і його вплив на надійність програмного забезпечення. Компонент, що самовідновлюється, може автоматично відновлюватися та повертатися до нормального стану, коли виникає збій. Оскільки конструкція компонента, що самовідновлюється, є дуже складною та дорогою, неможливо створити

самовідновлення для всіх компонентів. Таким чином, виявлення вузьких компонентів з метою їх самостійного відновлення на ранніх етапах розробки програмного забезпечення може мати великий вплив на надійність.

Нині було запропоновано кілька методів, заснованих на моделях проектування, для оцінки надійності та систем програмного забезпечення, але вплив самовідновлення на надійність, а також пошук компонентів, які мають великий вплив на надійність програмного забезпечення. Не було надано жодного звіту щодо самостійного відновлення компонентів на ранніх стадіях розробки програмного забезпечення. У роботі спочатку буде запропоновано метод моделювання самовідновлення за допомогою ланцюга Маркова, а потім будуть представлені чотири різні методи (без -рядів Тейлора - без самовідновлення, без рядів Тейлора - із самовідновленням, з - рядами Тейлора - без самовідновлення та з рядами Тейлора - із самовідновленням) для оцінки надійності програмної системи на основі її архітектури. Запропоновані співвідношення дозволять інженеру-програмісту визначити впливові та вузькі компоненти для самовідновлення.

Актуальність дослідження. Сучасні критичні програмні системи, такі як системи керування транспортом, промислові автоматизовані комплекси та медичне програмне забезпечення, потребують високого рівня надійності та безперервної роботи. Збої в таких системах можуть призводити до значних фінансових та людських втрат. Тому розробка механізмів самовідновлення є актуальним завданням, яке дозволяє зменшити ризики та підвищити стабільність функціонування програмних систем.

Практична значущість. Запропоновані методи можуть бути застосовані для підвищення надійності критично важливого програмного забезпечення в таких сферах, як транспорт, медицина, енергетика та фінансовий сектор. Використання механізмів самовідновлення дозволить зменшити витрати на обслуговування, знизити ризики відмов та забезпечити безперервність роботи програмних систем.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Стійкість і живучість в комп'ютерних мережах

Мережі загалом і глобальний Інтернет зокрема, стали важливими для повсякденної роботи. Споживачі використовують Інтернет для доступу до інформації, отримання продуктів і послуг, управляють фінансами та спілкуватися один з одним. Підприємства використовують Інтернет для ведення торгівлі зі споживачами та іншими підприємствами. Уряди залежать від мереж в їх щоденної роботі, наданні послуг, реагування на катастрофи тощо. Таким чином глобальний Інтернет описується як одна з критичних інфраструктур, від якої залежать наше життя.

Крім того, багато з цих інфраструктур мають залежності одна від одної [6]. Канонічний приклад, такий як живлення Інтернету залежить від електричної мережі, а електрична мережа все більше залежить від Інтернету SCADA (диспетчерське керування та збір даних).

Однак підвищена залежність від послуг роблять Інтернет більш вразливим до проблем. Із постійно зростаючою залежністю приходять два відповідні наслідки. По-перше, ця залежність призводить до збільшення наслідків збоїв. По-друге, посилення наслідків порушення призводять до того, що мережі стають все більш привабливими для кіберзлочинців. Стійкість слід розглядати як важливу конструкційну та експлуатаційну характеристику майбутніх мереж загалом, і глобального Інтернету зокрема. Уразливі місця Інтернету і потреби в більшій стійкості широко визнані [7]. Уданій роботі стійкість визначається як здатність мережі надавати і підтримувати прийнятний рівень обслуговування в умовах різноманітних несправності та проблеми з нормальною роботою.

Існує ряд відповідних дисциплін, які служать основою стійкості мережі. Оскільки ці дисципліни розвивалися незалежно протягом кількох

десятиліть, не існує усталеної самоузгодженої схеми та термінології. Опишемо ці дисципліни та в області стійкості після введення важливих понять несправність, помилка, ланцюг відмов (рисунок 1.1).

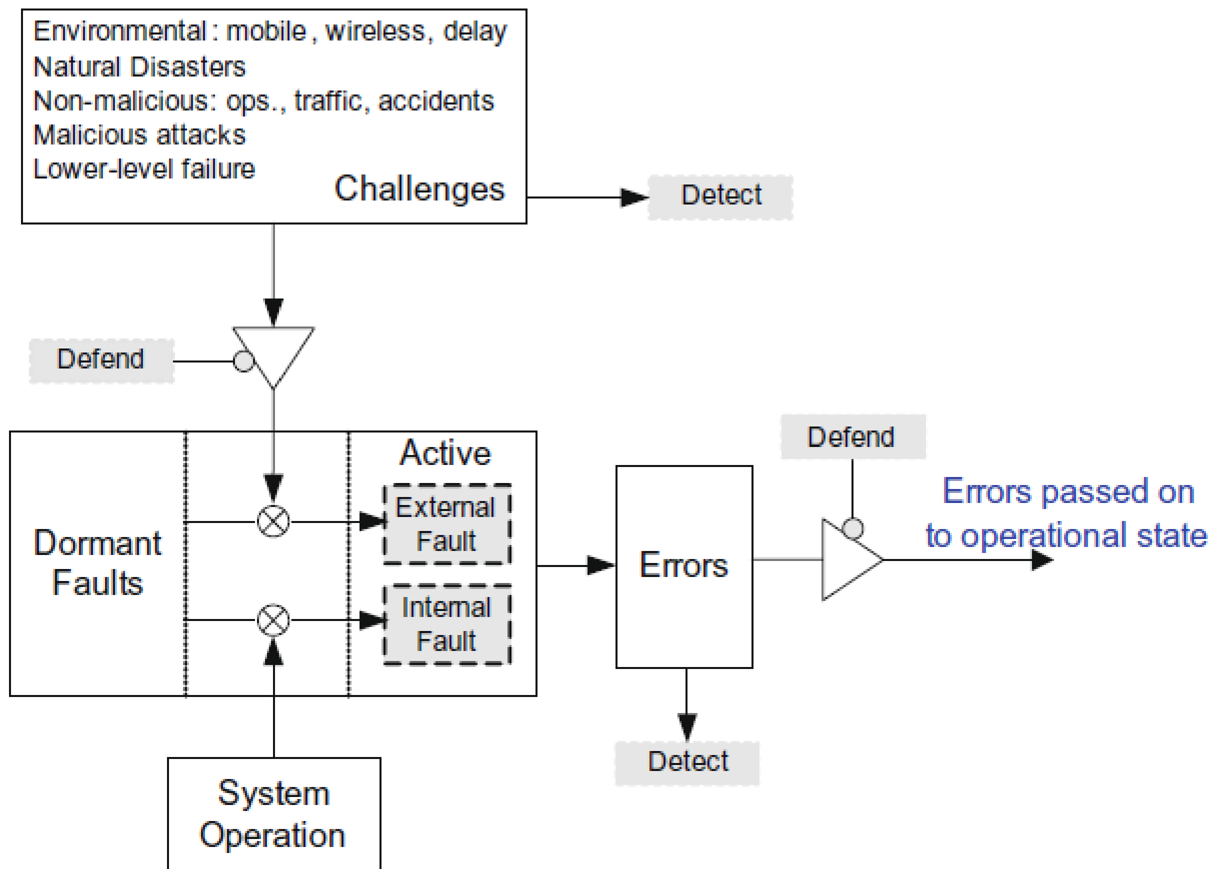

Рисунок 1.1 – Несправність, помилка, ланцюг відмов

Несправність — недолік у системі, який може спричинити помилку [8]. Це може бути або випадковий недолік конструкції (наприклад, помилка програмного забезпечення) або навмисна помилка через обмеження, які дозволяють зовнішньому виклику викликати помилку (наприклад, не розроблена достатньо міцна система через обмеження вартості). Може спрацювати неактивний збій, що призводить до активної несправності, яку можна спостерігати як помилку. Помилка - це відхилення від значення або стану програми, що може призвести збою. Збій служби (часто скорочується до збій) є відхиленням обслуговування від бажаного функціонування системи так, що вона відповідає специфікаціям або очікуванням [9].

Таким чином, помилка може бути викликати видиму помилку, яка може призвести до збою, якщо помилка проявляється таким чином, що система не відповідає специфікаціям. Цей зв'язок показано на рисунку 1.1. Захист мережі може запобігти появі несправності та багато іншим помітні помилкам та не призводить до збою. Порушення толерантності є одним із прикладів зменшення вплив несправностей і помилок на надання послуг. Крім того, можуть бути виявлені проблеми та помилки, які також забезпечують основу для дій, які вживаються як стратегічна частина стійкості.

На найвищому рівні можна виділити дві категорії, як показано на рисунку 1.2. Ліворуч розташовано завдання, що стосуються дизайну та інженерії систем. З правого боку – благонадійність дисципліни, які описують вимірювані властивості пружності системи. Відносини між цими двома формально є продуктивністю системи керування, а в нашому контексті – це надійність системи.

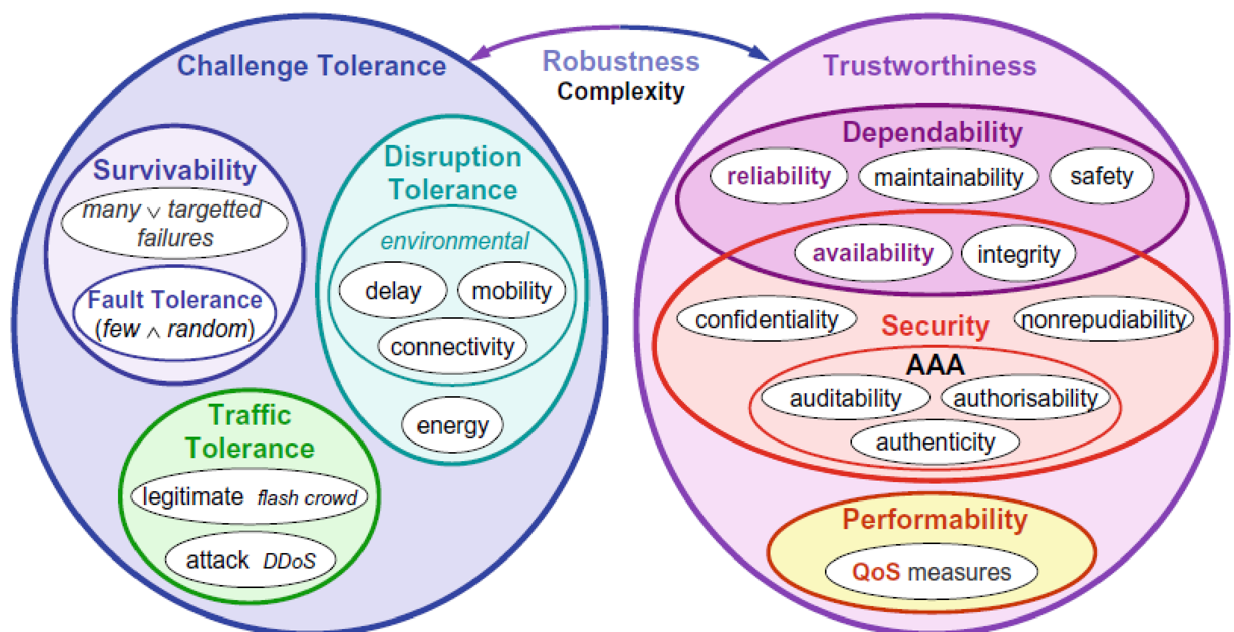

Рисунок 1.2 – Дисципліни стійкості

Перша основна підмножина стійкості дисциплін є угоди з проблемою того, як проектувати системи, щоб терпіти виклики, які перешкоджають

наданню бажаних послуг. Ці виклики можна розділити на компоненти і системні збої, відмовостійкість і живучість порушення комунікаційних шляхів, проблеми за рахунок збільшення трафіку в мережі. Наразі, захист мережі може запобігти викликам несправності та багато іншим помітним помилкам, які не призводять до збою. Порушення толерантності є одним із прикладів зменшення впливу несправностей і помилок на надання послуг. Крім того, можуть бути виявлені проблеми та помилки, які також забезпечують основу для дій, які вживаються частиною стратегії стійкості.

1.2 Методи оцінки надійності програмних систем

Оскільки наше повсякденне життя залежить від послуг програмних систем, методи оцінки надійності цих програмних систем мають велике значення. Вплив структури програмної системи на її надійність і коректність розглядається вже майже два десятиліття. Зросла присутність програмних систем в обладнанні, пристроях, послугах і повсякденній діяльності людей. Збої в комп'ютерних системах потрапляють у заголовки газет, оскільки вони спричиняють незручності для людей (вихід з ладу побутової техніки), економічний збиток (перебої в роботі банківських послуг) і, в крайніх випадках, смерть (збій систем управління польотом або медичного програмного забезпечення).

Важливо оцінити нефункціональні потреби на рівнях розробки програмного забезпечення. У компонентних системах такі нефункціональні вимоги, як надійність, ефективність і безпека, визначають якість кінцевого продукту та успіх програмного забезпечення [10]. Системи на основі компонентів утворюються з спільноти та зіставлення існуючих незалежних компонентів і взаємодіють одна з одною для надання послуг користувачам. Проблема оцінки нефункціональних потреб кожного з компонентів окремо є однією з галузей сучасних досліджень, і з іншого боку, навіть якщо

припустити, що компонент окремо має відповідну якість, відповідна якість поєднання компонентів один з одним не завжди гарантується. Тому на початкових етапах розробки, крім оцінки кожного компонента, необхідно оцінити, як вони взаємодіють один з одним.

Надійність програмного забезпечення визначається як імовірність того, що програмне забезпечення виконуватиме свої функції правильно (без збоїв) протягом певного періоду часу та за конкретних робочих умов і умов середовища, з якими воно стикається [11].

У літературі метод оцінки надійності програмної системи розглядався з різних точок зору, якими є (1) перспектива чорного ящика та (2) перспектива білого ящика [12]. Поширені підходи до оцінки надійності програмного забезпечення засновані на чорному ящику, тобто програмна система розглядається як єдине ціле, а моделюється лише її взаємодія із зовнішнім світом, без урахування внутрішньої структури. Три основні проблеми цих методів полягають у тому, що (1) оскільки вони недостатньо знають про внутрішнє функціонування системи програмного забезпечення, тому вони не можуть бути достатньо точними в оцінці надійності системи програмного забезпечення, (2) якщо після оцінки інженери програмного забезпечення приходять до висновку, що система не має належної надійності, заміна цього програмного забезпечення на нове програмне забезпечення або вирішення проблем поточного програмного забезпечення не є хорошим варіантом, оскільки це не буде економічно життєздатним, і (3) ці методи не можуть застосовувати на ранніх стадіях розробки програмного забезпечення та з моделей проектування.

На відміну від методів чорного ящика, методи білого ящика, оскільки вони знають внутрішню структуру програмної системи, можуть оцінити надійність програмної системи з прийнятною точністю. Основна перевага цих методів полягає в тому, що їх можна використовувати на ранніх стадіях розробки програмного забезпечення. Через те, що перевірка та оцінка цих функцій (функціональних і нефункціональних вимог) до етапу проектування

та впровадження витрачає менше часу та грошей, найкращим часом для оцінки поведінки системи є час, коли створюється архітектура цього програмного забезпечення. Архітектура програмного забезпечення, як перший продукт і результат етапу проектування програмного забезпечення, відіграє важливу і безпосередню роль у розробці складних програмних систем, і з її допомогою можна визначити оцінювану поведінку системи, тобто атрибути якості, як безпека, надійність, вимірювати зручність використання, змінність та ефективність.

Більшість методів білого ящика починають з оцінки та прогнозування надійності програмної системи на основі архітектури програмного забезпечення. Архітектура програмного забезпечення — це спосіб об'єднання основних компонентів програмної системи. Відповідно до стандарту IEEE, архітектура означає надання технічного опису системи, що показує структуру її компонентів, взаємозв'язок між ними, а також принципи та правила, що керують їх проектуванням та еволюцією з часом.

Отже, архітектура програмного забезпечення показує, як зібрані разом основні компоненти програмної системи. Кластеризація програмної системи є основною діяльністю пошуку відповідної архітектури. Насправді кластеризація класів програмного забезпечення – це процес групування класів програмного забезпечення таким чином, щоб класи з найвищим ступенем залежності поміщалися в кластер. Кластеризація полегшує розуміння програмного забезпечення та полегшує операції з обслуговування в майбутньому. Ця кластеризація здійснюється на основі зв'язків між класами. Загалом ці зв'язки показані у формі графіків залежності компонентів (наприклад, CDG), де вузли представляють програмні компоненти, а ребра моделюють зв'язки між ними.

Оцінка надійності з архітектури на ранніх стадіях виробництва програмного забезпечення відіграє значну та важливу роль у розробці програмних систем з високою надійністю. Розробка на основі компонентів вважається основним рішенням для подолання основних програмних

проблем. Тому дуже важливо надати модель для оцінки надійності програмного забезпечення на основі компонентів з точки зору архітектури. У літературі було представлено багато методів для опису архітектури програмного забезпечення для оцінки надійності, яка включає типи мереж Петрі (такі як HCPN, SPN) [13], ланцюги Маркова [14], байєсівські моделі.

Зрозуміло, що вплив компонентів на надійність програмної системи неоднаковий; Крім того, виконання компонента під час виконання програмної системи відрізняється від інших компонентів. Компонент із великою кількістю повторень має більший вплив на надійність програмної системи. Серед згаданих моделей лише ланцюги Маркова мають цю функцію, вони можуть обчислювати кількість разів виконання компонента під час виконання програмної системи. Тому в цій роботі ми використовували дискретний ланцюг Маркова для моделювання архітектури.

В останні роки багато наголошується на важливості додавання компонентам здатності до самовідновлення. Самовідновлювальні компоненти намагаються автоматично виявляти та виправляти помилки, які виникають під час їх використання. У цій роботі ми представимо метод моделювання компонентів самовідновлення за допомогою ланцюга Маркова, за допомогою якого ми зможемо оцінити надійність програмних систем з урахуванням властивості самовідновлення.

1.3 Ланцюги Маркова в оцінці надійності програмних систем

Розрахунок надійності програмної системи виконується на двох рівнях: компонентах та всієї програмної системи. На рівні компонента робиться спроба передбачити надійність компонента шляхом поєднання результатів аналізу внутрішньої структури та поведінки компонента з даними, отриманими в результаті випробувань або фактичного історичного досвіду. На рівні всієї програмної системи мета полягає в тому, щоб знайти надійність програмної системи на основі конфігурації та взаємодії між компонентами.

Існуючі моделі на основі архітектури поділяються на три широкі категорії: на основі стану, на основі шляху та адитивні. Моделі на основі станів (такі як ланцюги Маркова, мережі Петрі та автомати) використовують керуючі графи для представлення архітектури програмного забезпечення та використовують аналітичні методи для прогнозування надійності. Моделі на основі шляхів розраховують надійність програмного забезпечення відповідно до можливих шляхів виконання програми. Шляхи виконання можуть бути визначені за допомогою моделювання, виконання програми або алгоритму. Інкрементні моделі припускають, що надійність кожного компонента можна моделювати за допомогою неоднорідного процесу Пуассона (NHPP), який викликає процес відмови системи [20]. Серед трьох категорій моделей надійності програмного забезпечення на основі архітектури моделі на основі стану досліджувалися більше, ніж два інших методи.

У моделях на основі стану програмна архітектура може моделюватися DTMC, CTMC, SMP, DAG або SPN. DTMC, CTMC і SPM можна розділити на два типи: невідновлювані та абсорбуючі. SPN і DAG можна використовувати для моделювання одночасних програм. DAG обмежується моделюванням одночасних програм без циклів, але SPN також використовується для програм із циклами. Відмова компонента може бути показана надійністю компонента, постійною частотою відмов і залежною від часу серйозністю відмов. Методи оцінки надійності в підходах на основі стану поділяються на дві категорії: гібридні та ієрархічні, які показують, як розглядати поведінку відмов компонентів з архітектурою програмного забезпечення для прогнозування надійності. У комбінованому методі поведінка компонентів при відмові поєднується з архітектурою програми та отримується комбінована модель для прогнозування надійності системи. В ієрархічному методі спочатку архітектура програмного забезпечення моделюється моделями на основі стану, а потім надійність оцінюється за цією моделлю, враховуючи поведінку компонентів при збоях.

Ланцюг Маркова - це стохастичний процес без пам'яті. Стохастичні

процеси відносяться до явищ, результат яких невідомий до того, як вони відбудуться, наприклад, кидання монет або гральних кісток. Коли умовний розподіл ймовірностей для стану системи на наступному кроці залежить тільки від поточного стану системи і не залежить від попередніх станів, для моделювання використовується ланцюг Маркова. Поглинаючий ланцюг Маркова з дискретним часом є хорошим підходом для опису структури компонентів програмного забезпечення системи та дозволяє прогнозувати її надійність, і він має багато застосувань у реальному світі моделювання. Ланцюг Маркова - це послідовність злічених стохастичних змінних $\{X_n, n=0,1,2,3,\ldots\}$ з марковською властивістю наступним чином:

$$p\left\{X_{n+1}=j \middle| X_n=i, X_{n-1}=i_{n-1},\ldots,X_1=i_1,X_0=i_0\right\}=p\left\{X_{n+1}=j \middle| X_n=i\right\}=p_{ij}. \quad (1.1)$$

Серед ланцюгів Маркова з дискретним часом виділим дві категорії:

- нерозкладний: можливий доступ до кожного стану через усі інші його режими;
- поглинач: він має принаймні один стан, при досягненні якого, його стан більше не зміниться.

1.4 Життєвий цикл системи

Розглянемо конкретні принципи щодо самоадаптивних програмних систем, які є характеристикою їхнього життєвого циклу та підтримують життєвий цикл ефективними процесами, реалізуються програмним забезпеченням. Використовуючи самоадаптивні системи програмного забезпечення, розглянемо життєвий цикл типової програмної системи, точно вказуючи на явні та неявні зв’язки між процесами розробки програмного забезпечення та розгорнутою самоадаптивною програмною системою.

Суттєвою характеристикою самоадаптивної програмної системи є її

здатність автономно розвиватися та динамічно адаптувати свою поведінку у відповідь на зміни системних вимог, самої системи або операційного середовища системи.

Механізми розвитку та адаптації повинні зберігати суть поведінки системи, постійно забезпечуючи прийнятну реалізацію основних вимог системи. Головна увага полягає в тому, чи вводяться зміни в програмне забезпечення в автономному режимі чи в режимі онлайн. Щоб сформулювати механізми еволюції та адаптації, на рисунку 1.3 зображено концептуальну архітектуру самоадаптивних програмних систем. Важливою властивістю цієї архітектури є розподіл завдань, який сприяє поділу інтересів між модулями системи.

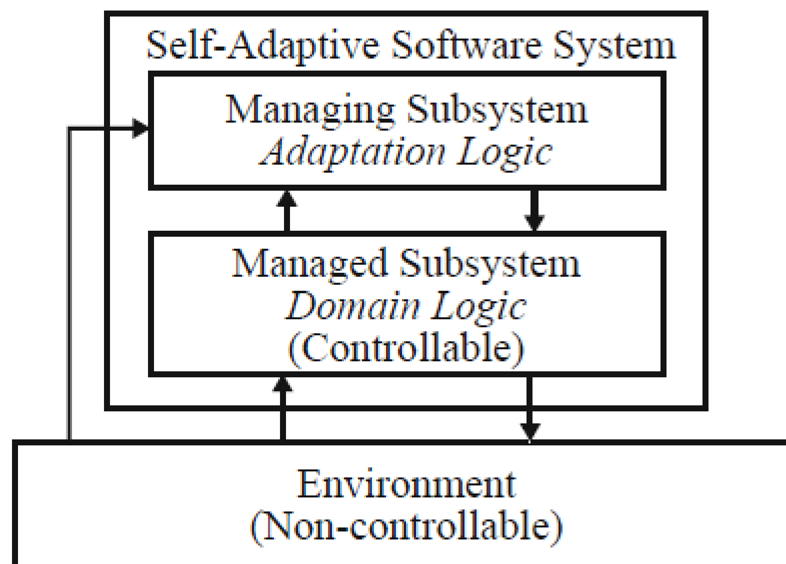

Рисунок 1.3 – Концептуальна архітектура для самоадаптивних програмних систем

Підсистема управління реалізує логіку адаптації. Усі функціональні задачі системи складають домени, кожен з яких підтримує виконання своєї групи задач управління. Логіка адаптації реалізує цикл керування відповідно до схеми монітор-аналіз-план-виконання, яка розвиває та адаптує логіку домену. Функціональність домену і основні вимоги системи реалізовані доменом логіки. Самоадаптивна система працює в неконтрольованому

середовищі, яке можна просто спостерігати за допомогою логіки адаптації, тоді як логіка домену може як спостерігати, так і впливати на навколишнє середовище. Крім того, архітектура допускає додаткові підсистеми керування, які адаптують логіку адаптації в інших підсистемах керування. Це може бути використано для опису, наприклад, рівня управління цілями в багаторівневій архітектурі або ієрархічного контролю в автономних обчисленнях. Ця характеристика самоадаптивних систем сприяє еволюції та адаптації механізмів, реалізованих логікою адаптації до хмарних обчислень, які мають підтримуватися пліч-о-пліч із логікою домену, поєднуючись із запущеною системою.

1.5 Постановка мети та завдань дослідження

Підсумуємо основні недоліки існуючих рішень.

Висока залежність від централізованих систем управління – традиційні методи забезпечення надійності програмних систем часто передбачають централізований контроль, що може спричинити збої в разі виходу з ладу головного вузла.

Обмежена швидкість реакції на відмови – більшість сучасних підходів до управління надійністю потребують часу для виявлення та усунення проблеми, що може призводити до тривалих простоїв системи.

Недостатня адаптивність до змін – існуючі рішення часто не враховують зміну робочих умов і не дозволяють швидко адаптувати систему до нових викликів.

Високі витрати на підтримку – традиційні методи резервування ресурсів та дублювання компонентів є дорогими та вимагають значних обчислювальних потужностей.

Метою роботи є підвищення надійності критичних програмних систем шляхом розробки та впровадження методів самовідновлення компонентів, що дозволяє зменшити кількість відмов та забезпечити безперервне

функціонування програмного забезпечення.

Завдання дослідження:

- аналіз існуючих підходів до забезпечення надійності програмного забезпечення;
- визначення критичних компонентів, що впливають на стабільність роботи програмних систем;
- розробка методики оцінки надійності програмного забезпечення з використанням ланцюгів Маркова;
- впровадження механізмів самовідновлення у критичних компонентах;
- експериментальна перевірка ефективності запропонованої методики.

Об'єкт дослідження: процеси забезпечення надійності в критичних програмних системах.

Предмет дослідження: методи та алгоритми самовідновлення програмних компонентів для підвищення стійкості програмного забезпечення.

2 МЕТОДИ САМОАДАПТАЦІЇ ТА САМОВІДНОВЛЕННЯ

2.1 Модифікований життєвий цикл програмного забезпечення з самоадаптацией

Автоматичні обхідні шляхи (AW) — це техніка, яка розширює програми за допомогою можливостей самоадаптації, які мають справу з функціональними збоями під час виконання [15].

Коли програми виходять з ладу через помилку в самій програмі або в одній із бібліотек, що використовуються програмою, техніка AW намагається замаскувати помилку й, таким чином, уникнути відповідних збоїв, забезпечуючи при цьому основні функції домену.

Техніка базується на гіпотезі про те, що програмні системи зазвичай пропонують кілька «еквівалентних операцій», які забезпечують однакові основні функції в різних реалізаціях. Якщо операція не вдається, механізм AW використовує цю внутрішню надлишковість, щоб автоматично знайти обхідні шляхи та застосувати альтернативу в еквівалентній послідовності операцій. Розглянемо, наприклад, компонент контейнера, який реалізує одну операцію для додавання одного елемента та одну операцію для додавання кількох елементів. Щоб додати два елементи, можна додати або один елемент за іншим, або додати їх обидва одночасно. Якщо один із цих варіантів викликає збій під час виконання, техніка AW намагається виконати еквівалентну послідовність операцій як альтернативний варіант маскування несправності.

Описана ситуація представлена на рисунку 2.1. Компонент програми викликає операцію, надану іншим компонентом (викликаючим і викликаним компонентом відповідно). Якщо виклик викликає збій, техніка AW, реалізована логікою адаптації, обробляє цей збій під час виконання, спочатку шукаючи послідовність операцій, яка еквівалентна невдалому виклику.

Знайшовши еквівалентну послідовність операцій, логіка адаптації виконує адаптацію, яка автоматично викликає цю послідовність операцій.

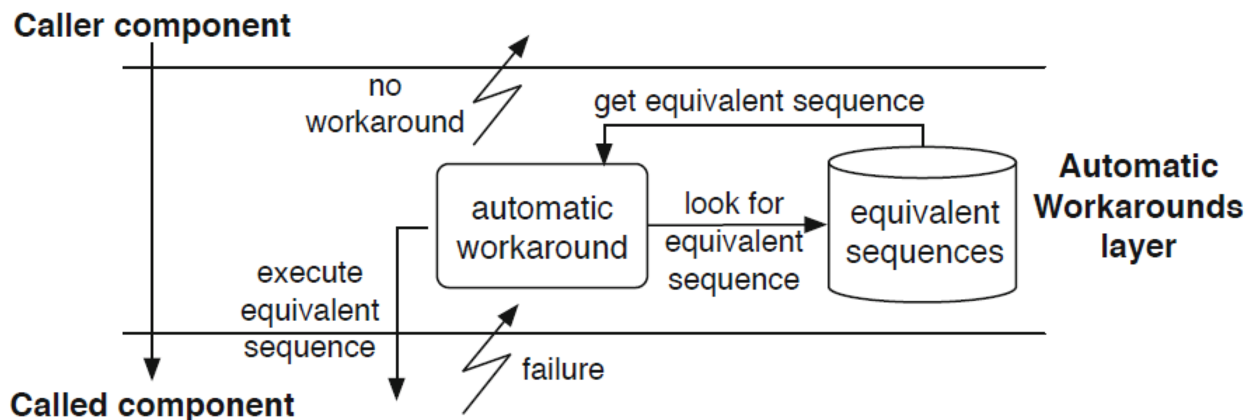

Рисунок 2.1 – Техніка автоматичного обходу

Якщо альтернативне виконання не спричиняє збій, це означає, що знайдено успішний обхідний шлях, і програма продовжує працювати так, ніби початкової помилки ніколи не було.

В іншому випадку логіка адаптації продовжує перевірку інших еквівалентних послідовностей до тих пір, поки альтернатива або успішно виконана, або поки всі еквівалентні послідовності не будуть протестовані безуспішно. В останньому випадку про збій повідомляється компоненту, що викликає задачу. Щоб усунути ці збої, розробники мають виправити помилку, виправивши несправний компонент або вручну визначивши та надавши дійсні обхідні шляхи для логіки AW adaptation.

У типовому програмному процесі звіт про помилку, заповнений користувачем, стане відправною точкою ручної роботи для вирішення проблем під час виконання. Розробникам потрібно буде виявити та проаналізувати основні причини проблеми, визначити та застосувати виправлення для помилки та, нарешті, розгорнути виправлення або повторно розгорнути всю програму. Техніка AW спрямована на автоматизацію та перенесення цих дій під час виконання до логіки адаптації. Це забезпечить своєчасне реагування без зупинки та повторного розгортання програми.

Як згадувалося вище, самоадаптивна програмна система виконує регулярні дії програмного процесу під час роботи системи. На рисунку 2.2 проілюстровано, як процес програмного забезпечення та його дії взаємодіють із працюючою самоадаптивною програмною системою.

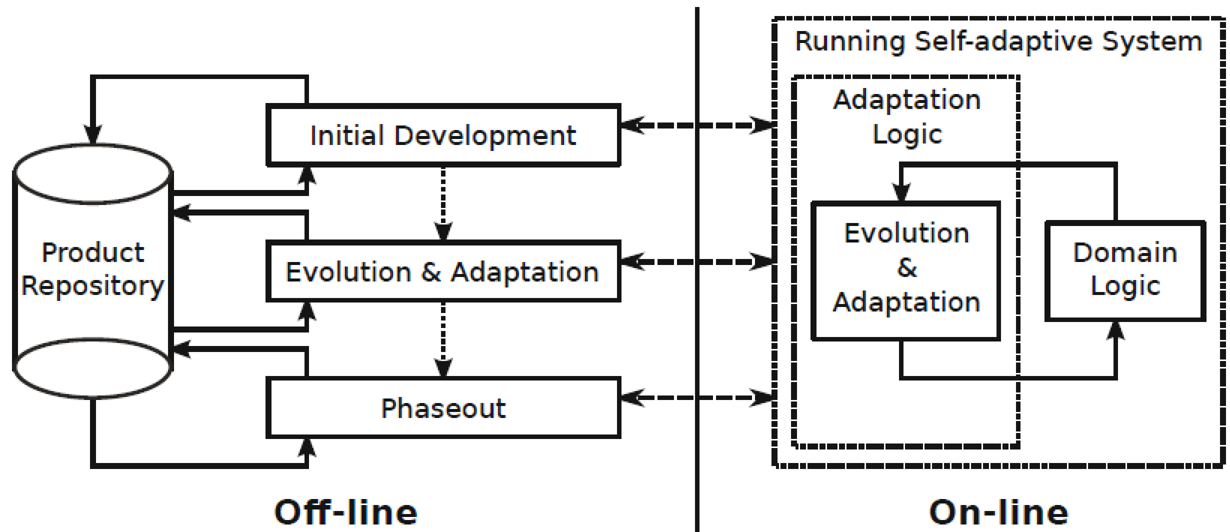

Рисунок 2.2 – Модель життєвого циклу для самоадаптивної програмної системи

Ліва частина малюнка зображує поетапну модель життєвого циклу. Етапи охоплюють початковий розвиток самоадаптивної системи, традиційну еволюцію та адаптаційні дії, що виконуються в автономному режимі. Дії в автономному режимі працюють над артефактами, такими як моделі дизайну або вихідний код, а не безпосередньо в запущеній системі. Останній етап, Phaseout, охоплює зупинку та виведення з експлуатації самоадаптивної системи. На перший погляд і зосередившись на лівій частині фігури, цей процес виглядає ідентично традиційному програмному процесу. Однак він взаємодіє з працюючою самоадаптивною системою. Ця взаємодія відбувається через онлайн-діяльність, пов’язану з еволюцією та адаптацією, що становить логіку адаптації самоадаптивної системи. Використання уявлень самоадаптивного під час виконання системи, діяльність в режимі онлайн розвивається та адаптує логіку домену або іншу логіку адаптації,

поки система функціонує для надання послуг (ілюстровано правою стороною рисунка 2.2). Взаємодії та залежності між офлайн і он-лайн діями, зображені двонаправленими стрілками, є специфічними для моделей життєвого циклу, націлених на самоадаптивні системи.

Щоб забезпечити більш глибокий аналіз взаємодії між процесом програмного забезпечення та працюючою самоадаптивною системою, потрібні більш детальний опис їх взаємодії. На рисунку 2.3 представлено екземпляр життєвого циклу для самоадаптивної системи, який використовує підхід автоматичного обхідного шляху (AW) для логіки адаптації та процесу її розробки. Перегляд часової шкали містить два графіки та точки взаємодії. Самий верхній графік відображає рівень активності процесі розробки (вісь y) та низку конкретних дій поза мережею протягом часу (вісь x). Рівень активності змінюється протягом життєвого циклу.

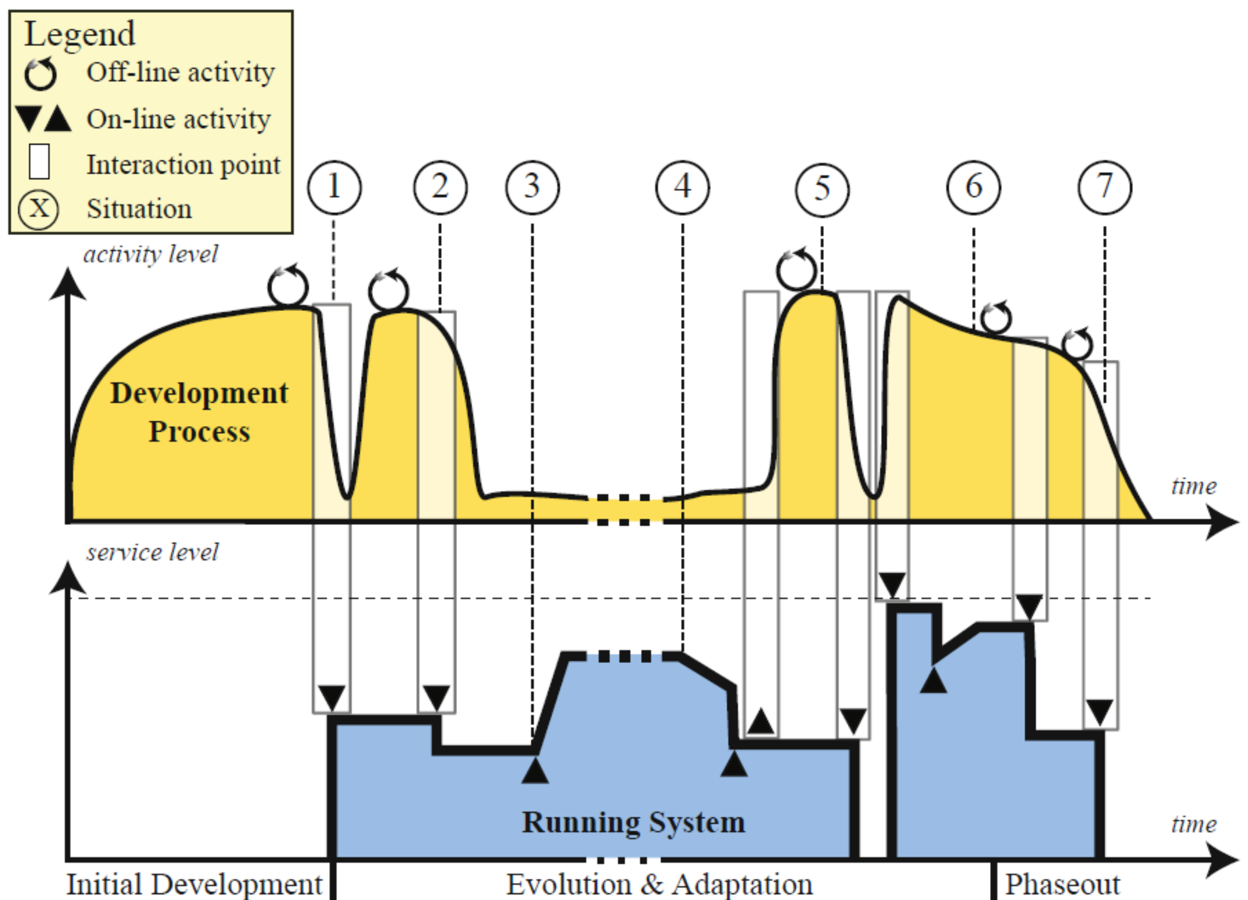

Рисунок 2.3 – Перегляд часової шкали процесу та самоадаптивної програми

Життєвий цикл поділяється на три чіткі стадії: початковий розвиток, еволюцію та адаптацію та поступову відмову.

Найнижній графік показує рівень обслуговування для самоадаптивної системи AW у часі (вісь x). Варіації на графіку зумовлені подіями, зовнішніми чи внутрішніми для системи. Наприклад, дії в режимі онлайн, ініційовані логікою адаптації або процесами розробки (діяльність з обслуговування та еволюції). Графік на рисунку 1.6 показує, що запущена система діє як зацікавлена сторона з певною роллю в процесі розробки, оскільки вона активно впливає на розробку та підтримку програмного забезпечення [2]. Однак у випадку самоадаптивних систем це також вірно для діяльності в режимі онлайн. Тепер ми можемо ідентифікувати та охарактеризувати низку сценаріїв, у яких взаємодіють офлайнові та онлайнові дії, як це показано позначеними ситуаціями рисунку 2.3.

Перший етап, початкова розробка, створює першу версію самоадаптивної системи програмного забезпечення шляхом ряду автономних операцій. У контексті підходу AW розробники програмного забезпечення розробляють логіку домену програми. Щоб покращити програму за допомогою техніки AW, вони мають надати початковий список еквівалентних послідовностей для операцій програми з відомими обхідними шляхами, як для офлайн-розробки. Це також є прикладом можливостей самоадаптації впливати на початковий розвиток. Після завершення початкової розробки система готова до розгортання. Початкове розгортання (див. ситуацію 1 на рисунку 2.3) запускає систему в роботу, що ілюструється кроком рівня обслуговування системи. Діяльність початкового розгортання фіксується першою точкою взаємодії. Точки взаємодії вказують на те, що діяльність поза мережею впливає на онлайн-діяльності або навпаки.

Коли екземпляр системи працює, починається етап еволюції та адаптації. Оскільки ми розглядаємо самоадаптивні програмні системи, адаптація та еволюція можуть бути ініційовані та контрольовані діяльністю в автономному режимі (процес), а також діяльністю в режимі онлайн (логіка

адаптації). Це проілюстровано шістьма додатковими ситуаціями після початкового розгортання.

Ситуація 2 ілюструє, як зміни в результаті офлайнової еволюції або адаптаційної діяльності згодом вводяться в працюючу систему за допомогою онлайнових дій. Оновлення в режимі онлайн розгортає нові або оновлені компоненти логіки домену. Для цих компонентів розробник має оновити списки еквівалентних послідовностей і, отже, логіку адаптації, щоб зберегти поведінку AW. У сценарії, який реалізується, рівень обслуговування системи зазнає негативного впливу після нового розгортання через ймовірні збої в нових компонентах.

Однак, якщо трапляються відповідні збої, вони обробляються AW technique, як показано в ситуації 3, завдяки попереднім спробам підтримувати список в автономному режимі еквівалентних послідовностей. Техніка AW відстежує збої в додатку та здатна впоратися з ними шляхом успішного застосування обхідних шляхів. Це переводить систему в стан з покращеним рівнем обслуговування. Ця ситуація є прикладом онлайн-адаптації, яка часто можлива завдяки попереднім діям поза мережею.

Тим не менш, техніка AW може не впоратися з довільними збоями, які постійно негативно впливають на рівень обслуговування системи (див. ситуацію 4). Це випадок, коли техніка AW не знаходить належного обхідного шляху (перша активність у мережі в ситуації 4), тобто всі доступні еквівалентні послідовності були протестовані безуспішно, і, як наслідок, помилка повторюється. У цьому випадку друга онлайн-активність AW у цій ситуації сповіщає розробників, які вводять автономний процес для усунення збоїв, наприклад, вручну виправлення несправності та підтримка списку еквівалентних послідовностей. Ця ситуація показує, як онлайн-діяльність взаємодіє з діяльністю поза мережею та запускає її. Якщо програму перероблено в автономному режимі, оновлення в режимі онлайн може бути надто складним, а тому неможливим. Такі радикальні зміни в системі фіксуються в ситуації 5 шляхом офлайнової еволюції з наступним

розгортанням. У цій ситуації запущена система вимикається, а новий випуск розгортається (подібно до ситуації 1), що впливає на доступність системи та, отже, на рівень обслуговування. Ситуація 6 висвітлює випадок, коли офлайн-діяльність розвивається або адаптується в режимі онлайн (логіка адаптації) з подальшим внесенням цих змін до запущеної системи. У контексті логіки адаптації AW у будь-який момент часу розробники можуть ідентифікувати та вказати нові еквівалентні послідовності, що є автономною діяльністю, яка налаштовує механізм AW. Завдяки онлайн-діяльності ці нові послідовності вводяться в знання AW, і ці послідовності можуть бути використані в наступних адаптаціях логіки домену. Нарешті, ситуація 7 ілюструє повну зупинку та виведення з експлуатації як частину етапу поступового виведення з експлуатації, оскільки було прийнято рішення припинити роботу системи. Відключення та виведення з експлуатації, які плануються та ініціюються в автономному режимі, завершують життєвий цикл системи та з деякою затримкою життєвий цикл процесу.

Діяльність з еволюції та адаптації, що виконується між точками взаємодії, зазвичай здійснюється в режимі он-лайн, якщо ними керує логіка адаптації. Навпаки, вони виконуються в автономному режимі, якщо ними управляє процес, керований людиною. У цьому контексті точки взаємодії синхронізують дії або артефакти в режимі офлайн і онлайн. Як приклад, ситуація 6 ілюструє, що дії в режимі онлайн можна розвинути та адаптувати в режимі офлайн, а подальше динамічне оновлення синхронізує ці зміни в режимі офлайн із відповідними діями в режимі онлайн у логікі адаптації.

2.2 Особливості пропонованого методу

Загальна мета – розрахувати надійність програмної системи, враховуючи властивість самовідновлення найвпливовіших компонентів. Блок-схема запропонованого підходу показана на рисунку 2.4. Деталі кожної з частин цієї блок-схеми описані нижче.

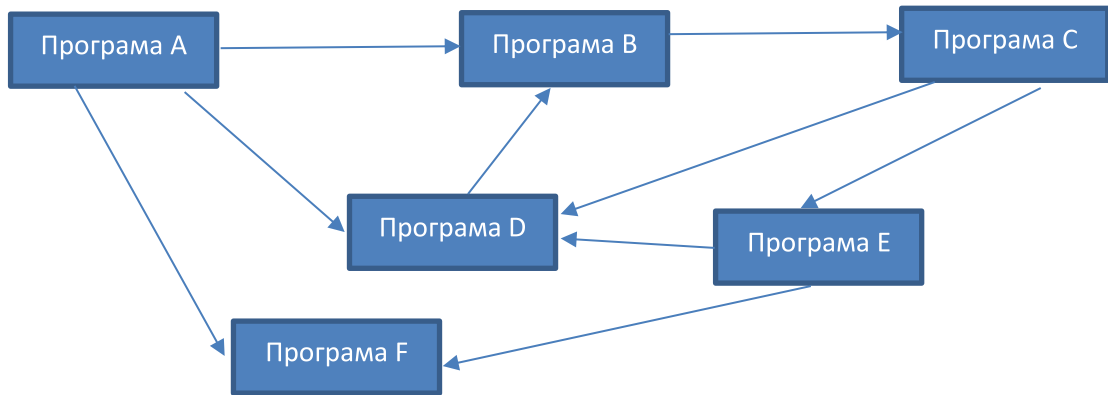

Рисунок 2.4 – Приклад архітектури програмної системи для задачі комівояжера

Етап 1: спочатку ми створюємо архітектуру програмного забезпечення за допомогою інструменту Bunch і перетворюємо цю архітектуру на дискретний ланцюг Маркова з поглинанням часу та обчислюємо матрицю ймовірності переходу, фундаментальну матрицю та матрицю дисперсії. Створення дискретного ланцюга Маркова з архітектури програмного забезпечення виглядає наступним чином:

- кожен компонент в архітектурі буде еквівалентний кожному стану в ланцюзі Маркова;
- нехай F_{in} і F_{out} представляють кількість викликів між двома компонентами x і y і кількість вихідних викликів від компонента x, відповідно, в архітектурі. Імовірність переходу між x і y визначається рівнянням F_{in} / F_{out} .

Для програмної системи, що включає кілька кластерів, ми можемо показати структуру програмної системи (архітектуру програмного забезпечення) за допомогою ланцюжка Маркова. Стани ланцюга Маркова представляють кластери, а ребра між станами представляють передачу управління від одного кластера до іншого. Наприклад, розглянемо відому задачу комівояжера (TSP). Спочатку ми витягуємо структуру TSP з вихідного коду за допомогою інструменту Bunch. Входом інструменту Bunch є граф викликів, а його виходом є структура програмного забезпечення (архітектура

програмного забезпечення). Комерційний інструмент NDepend [16] використовувався для вилучення графа викликів TSP з його вихідного коду. Інструмент NDepend для більшості відомих у світі мов програмування може витягувати граф викликів із вихідного коду. Після вилучення графа викликів його слід кластеризувати, щоб отримати відповідну архітектуру. На рисунку 2.4 показана витягнута архітектура для проблеми TSP з її вихідного коду.

Після вилучення архітектури ми перетворюємо їх у ланцюги Маркова. Ланцюг Маркова (рисунк 2.5).

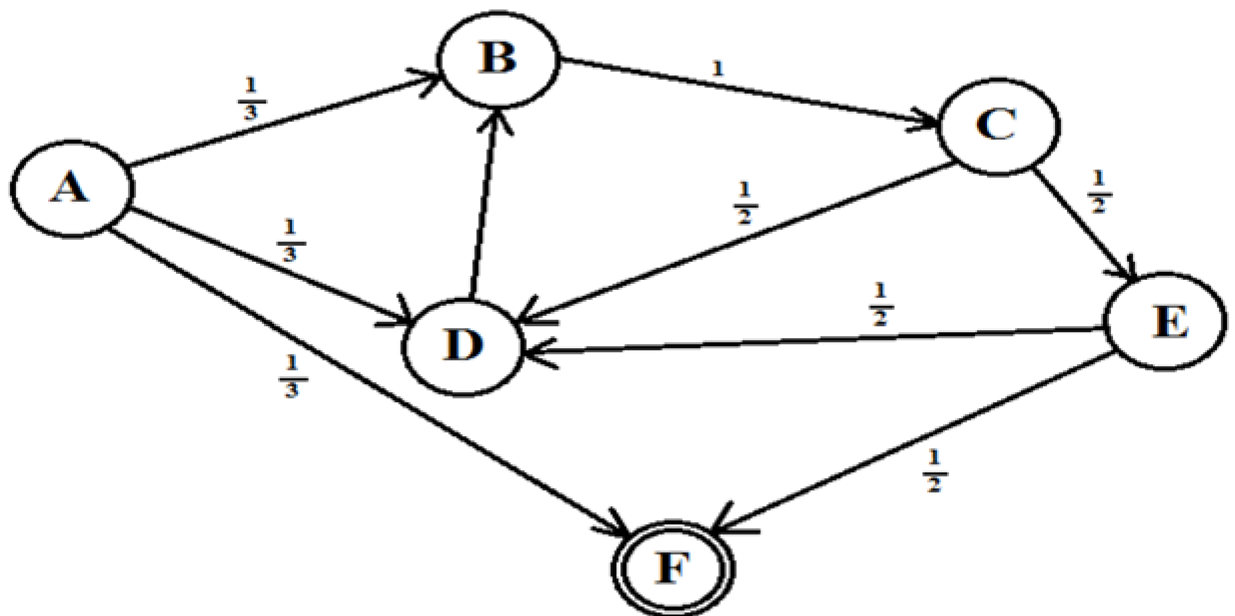

Рисунок 2.5 – Ланцюг Маркова (для системи на рисунку 2.4)

Цифри по краях вказують на ймовірність переходу з одного кластера в інший. Наприклад, на рисунку 2.4 з кластера 3 виходить загалом 3 ребра, 1 з яких пішов у кластер 4, 1 — у кластер 5, а 1 ребро — у кластер 6. Кластер 6 додається як кінцевий стан до ланцюга Маркова, який зазвичай не представлений в архітектурі. Кожен з A, B, C, D і E є кластерами, які містять принаймні один або більше компонентів у собі, кластер F є кінцевим станом у ланцюзі Маркова, який не має вихідного краю та поглинається.

Загалом, пропоновані етапи методу представлені на рисунку 2.6.

Рисунок 2.6 – Пропоновані етапи процесу

Етап 2: Частота ремонту та частота відмов: на цьому етапі ми пропонуємо метод розрахунку ефекту самовідновлення компонентів. Компонентна програмна система складається з комбінації кількох компонентів, і кожен компонент має частоту відмов і частоту ремонту. Оскільки надійність компонента покращується завдяки властивостям самовідновлення, тому необхідно розрахувати надійність компонента самовідновлення. Нехай m_i і n_i позначають інтенсивність відмов і інтенсивність ремонту кожного компонента відповідно. На початку компоненти справні та виконують свої завдання належним чином, через помилки програмування чи з інших причин програмна система може вийти з ладу. Залежно від його функціональних можливостей, система може вийти з ладу, якщо один або кілька компонентів системи вийдуть з ладу. Таким чином, для кожного компонента швидкість ремонту вважається ймовірністю того, що компонент повернеться в правильний стан у разі поломки (тобто компонент може самовідновити себе і продовжувати працювати). Іншими словами, коли компонент перебуває в стані збою, він, швидше за все, буде відремонтований і повернеться в безпечний стан.

У програмній системі режим компонента в наданні послуг залежить від поточного стану, інтенсивності відмов і інтенсивності самовідновлення компонента та не залежить від часу та станів компонента в попередніх посиланнях, тому є випадковим процесом. Його можна моделювати за допомогою ланцюга Маркова з дискретним часом.

Щодо визначення надійності компонента, що самовідновлюється, ми повинні знати, що, починаючи з безпечного стану компонента, чому дорівнює ймовірність посилань на компонент у безпечному стані (вираз 1.1).

$$P = \begin{vmatrix} p_{00} & p_{01} \\ p_{10} & p_{11} \end{vmatrix} = \begin{vmatrix} 1-m_i & m_i \\ n_i & 1-n_i \end{vmatrix}. \quad (2.2)$$

Ланцюг Маркова для матриці P показано на малюнку 2.7.

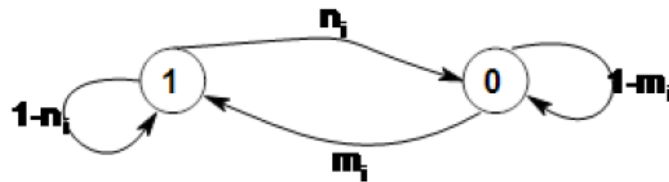

Рисунок 2.7 – Однокрокова матриця ймовірностей ланцюга Маркова для компонентів самовідновлення

Розрахунок інтенсивності самовідновлення та інтенсивності відмов: згідно з положенням моделі ланцюга Маркова, для незвідного ланцюга Маркова зі скінченною кількістю станів стабільний стан є унікальним. Відповідно до матриці переходу та використовуючи рівняння, що стосуються розподілу стійкого стану, можна вирахувати значення щодо рівнянь марковської моделі для кожного компонента.

Етап 4. На цьому кроці пропонуються чотири метрики, що враховують різні комбінації рядів Тейлора та самовідновлення, щоб оцінити надійність програмної системи за її архітектурою. Нехай R_i позначає надійність

компонента i в програмній системі. Тоді загальна надійність цієї системи розраховується наступним чином:

$$R = \prod_{i=1}^{n} R_i . \quad (2.3)$$

Оскільки компоненти з великою кількістю повторень під час типового запуску мають великий вплив на надійність програмної системи, щоб підвищити точність, враховуємо кількість повторень кожного компонента. Нехай $m_{1,i}$ позначає очікувану кількість відвідувань компонента i, починаючи з першого компонента. Таким чином, загальна надійність програмної системи буде наступною:

$$R = \prod_{i=1}^{n} R_i^{m_{1,i}} . \quad (2.4)$$

Враховуючи, що використовується статична структура програмного забезпечення для прогнозування його надійності, і неможливо точно визначити надійність кожного компонента та кількість повторень кожного компонента на етапі проектування, щоб зменшити похибку в оцінці надійності, ми використовуємо апроксимацію рядів Тейлора другого та третього порядку. Нехай R_i, $(1, i)$, m_i та n_i відповідно вказують на надійність компонента i, очікувану дисперсію відвідування компонента i, швидкість відновлення компонента і та частоту відмов компонента i.

Рівняння 2.4 – 2-8 представляють показники для розрахунку надійності з урахуванням наступних комбінованих режимів: без ряду Тейлора - без самовідновлення (вираз 2.4), з рядом Тейлора другого порядку - без самовідновлення (вираз 2.5), з рядом Тейлора другого порядку - із самовідновленням, де виконаємо заміну $R_i = \frac{m_i}{m_i + n_i}$ (вираз 2.7):

$$R = \prod_{i=1}^{n} \left| R_i^{m_{1,i}} + \frac{1}{2} \left(R_i^{m_{1,i}} \right) \cdot \left(\log R_i \right)^2 \cdot \sigma_{1,i}^2 \right|. \tag{2.5}$$

$$R = \prod_{i=1}^{n} \left| R_i^{m_{1,i}} + \frac{1}{2} \left(R_i^{m_{1,i}} \right) \cdot \left(\log R_i \right)^2 \cdot \sigma_{1,i}^2 \right|. \tag{2.6}$$

$$R = \prod_{i=1}^{n} \left| \left(\frac{m_i}{m_i + n_i} \right)^{m_{1,i}} + \frac{1}{2} \left(\left(\frac{m_i}{m_i + n_i} \right)^{m_{1,i}} \right) \cdot \left(\log \left(\frac{m_i}{m_i + n_i} \right) \right)^2 \cdot \sigma_{1,i}^2 \right|. \tag{2.7}$$

Таким же чином можна отримати вирази для ряду Тейлора третього та більших порядків.

Етап 4. На даному етапі проводиться аналіз чутливості: на основі метрик, наведених на кроці 3, на цьому кроці для розрахунку впливу надійності кожного компонента на надійність усієї програмної системи представлено чотири метрики. Аналіз чутливості використовується для виявлення вузьких компонентів. На цьому етапі обчислюються наступні пераметри: чутливість впливу на надійність компонента без серії Тейлора без самовідновлення; чутливість впливу на надійність компонента без ряду Тейлора другого порядку з самовідновленням; чутливість вплив на надійність компонента без ряду Тейлора з самовідновленням; чутливість впливу на надійність компонента з рядом Тейлора другого порядку з самовідновленням.

3 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ

3.1 Моделювання процесів для самоадаптивних програмних систем

У цьому розділі обговорюється структура фрейворка, який створено для моделювання процесу, який може допомогти в розробці самоадаптивних програмних систем. Він вимагає, щоб мови моделювання процесів підтримували нові концепції, такі як он-лайн і офлайн дії. Таким чином, моделювання процесів також допомагає осягнути та зрозуміти реконцептуалізацію процесів, оскільки моделі процесів роблять ці концепції явними. Крім того, обговорюється низка дослідницьких проблем, які залишилися, в першу чергу пов’язаних із розробкою та застосуванням мови моделювання для розробки самоадаптивних програмних систем. Загалом процес – це механізм систематичного досягнення мети, як-от дотримання інструкцій зі складання продукту або дотримання офісних процедур. Подібним чином виконання інженерної діяльності з розробки та розвитку програмного продукту є процесом. Те, як виконується процес, визначається моделлю процесу, яка визначає частково впорядкований набір того, що робиться, коли, де і ким. Таким чином, інструкції зі складання, офісні процедури або опис діяльності програмної інженерії є моделями процесів, і такі моделі матеріалізують відповідні процеси. Ця матеріалізація доповнює людську роботу, підтримує координацію та комунікаці, і це підтримує аналіз, покращення, повторне використання, виконання або взагалі управління процесами [15,17].

Щоб використовувати такі переваги моделювання процесів у сфері програмної інженерії, було запропоновано кілька мов моделювання для опису програмних процесів [16]. Одним із прикладів таких мов моделювання є специфікація мета-моделі програмного забезпечення та системного процесу (SPEM) [18]. SPEM в роботі використовувався для ілюстрації обговорюваних

концепцій завдяки його гнучкості, розширюваності та придатності для розробки на основі моделей. Специфікація SPEM явно визначає мову моделювання за допомогою метамоделі для опису процесів розробки програмного забезпечення. Маючи чітке визначення мови моделювання, можна обговорювати та розширювати мову. Як обговорювалося та демонструвалося нижче, розширення мови потрібне для звернення до поняття, які походять від реконцептуалізації процесів для випадку самоадаптивних систем. Не відомо про жодну мову моделювання процесів, яка б підтримувала ці концепції як першокласні елементи та забезпечувала сувору основу для інженерних підходів на основі моделі.

Визначаючи метамодель, SPEM надає мову моделювання для визначення методів і процесів розробки програмного забезпечення, а також пропонує початкову підтримку для налаштування та впровадження процесів у конкретних проектах. Однак мова є загальною, оскільки вона призначена для підтримки моделювання процесів, які охоплюють проекти від моделі водоспаду до гнучких підходів. Таким чином, мова підтримує лише основні та абстрактні поняття, які присутні в будь-якому підході до розробки. Ці абстрактні поняття зображені на рисунку 3.1, який демонструє концептуальний і частковий погляд на мета-модель SPEM.

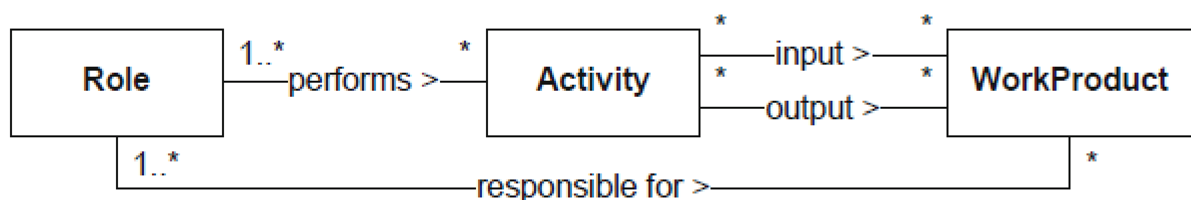

Рисунок 3.1 – Концептуальний погляд на мета-модель SPEM

Використовуючи цю метамодель, можна описати робочий процес дій, які виконуються ролями та мають вхідні або вихідні дані. Діяльність може бути пов’язана з іншою діяльністю, передаючи данні разом із діяльністю, тобто результат однієї діяльності є входом для іншої діяльності. Обов’язки

для роботи можна розподілити по ролях. Крім того, мова SPEM надає кілька елементів для представлення фаз, ітерацій і етапів. Нарешті, SPEM дозволяє вказувати різні форми залежностей, наприклад, між діяльністю або між робочими продуктами, насамперед для охоплення зв’язків між діяльністю або зв’язків складу та впливу між робочими продуктами. Однак ми не описуємо далі такі просунуті елементи, оскільки вони не є критичними для конкретних завдань, які розглядаються в цій статті.

Нижче обговорюєтеся, як було розширені основні поняття, визначені мовою SPEM, додатковими поняттями, які походять від переконцептуалізації програмних процесів для конкретного випадку самоадаптивних систем. Це робить додаткові концепції явними в моделях процесів, і це допомагає вирішувати проблеми в розробці самоадаптивних програмних систем.

3.2 Моделювання процесу самовідновлення на основі SPEM

Як обговорювалося вище, реконцептуалізація процесів програмного забезпечення для моделювання самоадаптивних систем вимагає нового виміру для класифікації діяльності. Цей вимір дозволяє розрізнити ситуації онлайн і офлайн, що, однак, вимагає, щоб залежності між ситуаціями онлайн і офлайн були явними та керованими. Крім того, витрати та переваги альтернативних он-лайн і автономій діяльності необхідно враховувати, оскільки вона керує розробкою та впливає на дизайн процесів і систем. Тому, як показано на рисунку 3.2, основну метамодель була розширена та визначена у SPEM, додатковими концепціями. У рамках роботи ці розширення, а також мета-моделі слід розглядати на концептуальному рівні, а не на технічному рівні як остаточну та повністю визначену мову моделювання. Таким чином, ми не обговорюємо, як розширення можуть бути найкраще реалізовані або реалізовані в рамках повної мета-моделі, визначеної в специфікації SPEM [18].

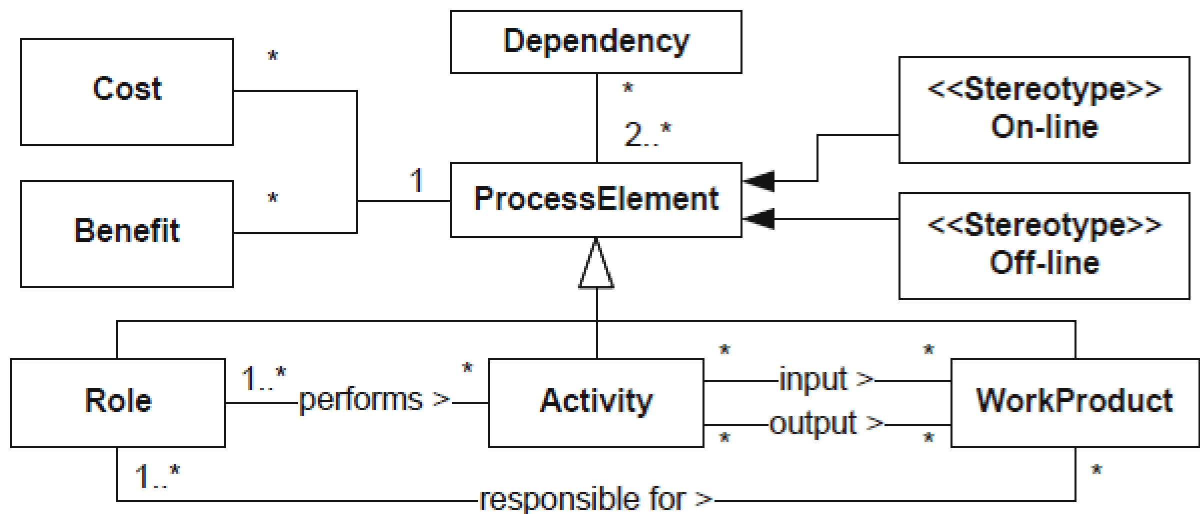

Рисунок 3.2 – Розширена мета-модель SPEM для процесів програмних систем з самовідновленням

Щоб зберегти розширення оригінальної мета-моделі (рисунок 3.1) простими та загальними, ми додали ProcessElement як загальний супермета-клас для мета-класів ролей, діяльності та робочих продуктів. Усі подальші розширення посилаються на цей ProcessElement і, таким чином, вони посилаються на всі три ролі: концепції, діяльності та роботи. Перш за все, стереотипи On-line і Off-line були визначені для елементів процесу, щоб чітко визначити, чи обслуговується будь-який елемент процесу в режимі онлайн чи офлайн.

Точніше, якщо стереотип асоціюється з роллю, він вказує на те, чи є роль частиною самоадаптивної системи (он-лайн) чи ні (оф-лайн). Для робочого продукту це вказує, чи такий робочий продукт виробляється або зазвичай використовується онлайн чи офлайн. Подібним чином, застосовуючи стереотипи до діяльності, моделі процесу можуть чітко розрізнити, чи виконується будь-яка діяльність онлайн чи офлайн.

Крім того, ми розширили мову моделювання SPEM концепцією залежності, яка пов’язує два або більше довільних елементів процесу. Це поняття залежності є більш піддатливим і гнучким для концептуальних дискусій, ніж конкретні можливості, надані SPEM для охоплення, наприклад, залежності між видами діяльності або між робочими продуктами. Однак, щоб

реалізувати ці розширення в рамках мета-моделі SPEM, слід розглянути вже існуючі засоби визначення залежностей. Надання загального поняття таких концепцій робить довільні залежності, такі як різні форми взаємодії між діяльністю онлайн і офлайн, показані в розділі 2, явними в моделях процесів.

Нарешті, витрати та вигоди можуть бути пов’язані з будь-яким елементом процесу. Це підтримує проектні рішення, що стосуються обсягу діяльності в режимі онлайн і поза мережею, і має дати відповіді на такі запитання, як: «Які витрати та переваги виконання цієї діяльності в режимі онлайн, на відміну від виконання її в режимі офлайн, і які інші види діяльності впливають або навіть потрібні для варіантів в режимі онлайн і поза мережею?»

Наведемо приклад того, як розширену мову SPEM можна використовувати для моделювання процесу програми, яка покладається на підхід AW для досягнення самоадаптації. На рисунку 3.3 наведено високорівневий структурний вигляд підходу. Це подання включає всі основні поняття мови SPEM, якими є ролі, дії та робочі продукти.

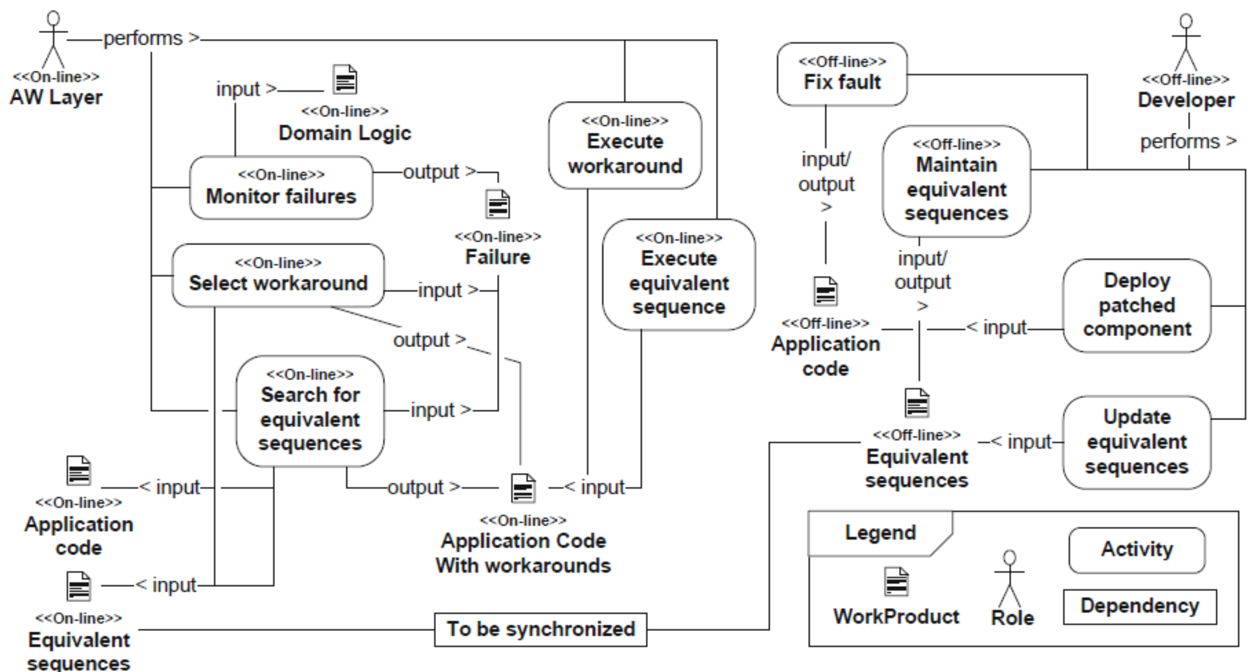

Рисунок 3.3 – Ролі, дії та робочі продукти у підході автоматичного обходу

Ролі зображуються піктограмами акторів, дії – округленими прямокутниками, а робочі продукти – артефактами документів. Як обговорювалося вище, ці елементи процесу можуть мати стереотипне значення Off-line або On-line, щоб позначити, чи належать вони відповідно до офлайнової чи онлайнової частини процесу. Нарешті, залежності представлені прямокутниками, з'єднаними з взаємозалежними елементами процесу.

Як показано на рисунку 3.3, у підході AW є дві ролі: рівень AW і розробник. Обидві ролі виконують дії, які мають робочі продукти як вхідні або вихідні дані. Рівень AW як логіка адаптації є частиною працюючої самоадаптивної системи, і, отже, виконує роль в режимі онлайн. Він контролює виконання програми, тобто логіки домену, в режимі онлайн. Якщо трапляється збій і його було виявлено, рівень AW відповідає або за вибір обхідного шляху, якщо він відомий із попередніх виконання, або за пошук еквівалентних послідовностей, якщо обхідний шлях невідомий. Таким чином, код програми з обхідними шляхами вибирається або створюється шляхом інтеграції перспективних еквівалентних послідовностей у код програми логіки домену. Нарешті, рівень AW виконує скоригований код програми і, таким чином, адаптацію, виконуючи відомий обхідний шлях або еквівалентні послідовності в спробі знайти дійсний обхідний шлях.

Розробник – це автономна роль, яка підтримує список еквівалентних послідовностей для компонентів логіки домену. Якщо AW layer не вдається автоматично знайти дійсний обхідний шлях, розробник виправляє відповідні помилки в несправних компонентах і розгортає виправлений компонент у працюючій системі. Це вимагає, щоб підтримуваний список еквівалентних послідовностей для цього компонента оновлювався на рівні AW для майбутнього використання в режимі онлайн. Це оновлення синхронізує список еквівалентних послідовностей, які розробник підтримує в автономному режимі, зі списком еквівалентних послідовностей, які використовуються в режимі онлайн на рівні AW. Це вирішує залежність "to

be synchronized" між цими двома робочими продуктами. Замість того, щоб приховувати такі залежності в діяльності, їх слід зробити явними в моделях процесів. Інакше вони можуть загубитися, коли процес і його діяльність змінюються або еволюціонують.

3.3 Опис результатів проведення експериментів

Завдяки фреймворку моделювання можна перенести фокус на проектування, прийняття рішень і обговорення. Однією з ключових проблем, які було визначено у розробці самоадаптивних систем, є ефективний розподіл процесів між діяльністю офлайн і он-лайн. Як згадувалося в попередньому розділі, ці дії можуть мати витрати та вигоди, пов’язані з тим, щоб допомогти у визначенні процесу для кожної конкретної самоадаптивної системи, яка приносить користь її зацікавленим сторонам. Значення тут має широке значення, яке повинно охоплювати низку цілей: цілі системи (кількісні та якісні), невизначеність, яка характеризує середовище виконання (що визначає масштаб адаптації), обмеження ресурсів середовища виконання (для підтримки онлайн-дій) і доступність до віддалених ресурсів (для підтримки автономної діяльності). Це вимагає кількісних характеристик на рівні визначення процесу, які належним чином доповнюються стохастичними характеристиками, щоб належним чином враховувати аспект невизначеності проблеми.

Як ілюстрацію принципів і практик у такому підході, можна використати програмну інженерію на основі цінностей (VBSE). VBSE підтримує кращі рішення щодо розробки програмного забезпечення, забезпечуючи економічну перспективу, де цінність поєднує поділ проблем, які використовуються для управління складністю, таким чином дозволяючи досягти глобальних оптимумів. У результаті процесу розробки програмного забезпечення, система програмного забезпечення, має низку пов’язаних цілей. Метою процесу розробки є отримання рішення, яке оптимізує цінність

(пов’язану з цілями) продукту в поточних умовах. Інженерні процеси характеризуються передбачуваними результатами, тобто рішення, прийняті в процесі, мають добре відомі наслідки. Іншою характеристикою є безперервний пошук альтернативних рішень і вичерпна оцінка альтернативних рішень для забезпечення достатніх знань, на яких ґрунтуватимуться рішення.

VBSE зосереджена навколо теорії, яка спрямована на те, щоб усі зацікавлені сторони проекту стали переможцями. VBSE пропонує чотири допоміжні теорії для «досягнення та підтримки взаємовигідного стану»: теорія залежності, теорія корисності, теорія прийняття рішень і теорія контролю. Теорія W і VBSE розроблені з моделлю процесу, яка підтримує розділи між процесами програмного забезпечення та запущеною системою. Наша гіпотеза полягає в тому, що реконцептуалізація, розглянута вище, фундаментально вплине на VBSE. Однак запропонований підхід, коли діяльність моделюється в єдиний спосіб, прокладає шлях для налаштування VBSE для розробки самоадаптивних програмних систем.

На додаток до структурних аспектів, як показано на рисунку 3.3, поведінка процесу також повинна бути визначена. Оригінальна мова SPEM дозволяє інтегрувати зовнішні мови для моделювання поведінки, такі як діаграми активності UML.

Подібним чином, розширена мова SPEM, яка використовувалась, не визначає власний формалізм моделювання поведінки, а використовує діаграми активності UML. Для нашого підходу AW на рисунку 3.4 зображено діаграму активності UML, яка представляє робочий процес дій у випадку, коли виникає збій і його потрібно вирішити.

Діаграма активності представляє етап еволюції та адаптації в часовій шкалі процесу (рисунок 2.3) для системи, яка спирається на підхід AW. Модель складається з двох розділів, по одному для кожної ролі, а саме рівня AW і розробника. Кожен розділ містить дії, що виконуються відповідною роллю, і модель визначає робочий процес дій усередині та між розділами

відповідно ролей. Ролі та види діяльності, які використовуються на діаграмі діяльності, такі ж, як і в поданні структурного процесу, зображеному на рисунку 3.3.

Рівень AW відстежує стан програми для виявлення збоїв. Коли виникає помилка, він вибирає обхідний шлях, якщо такий уже доступний у попередніх виконаннях. Якщо обхідний шлях доступний, AW layer виконує його негайно онлайн.

Якщо обхідний шлях невідомий, тоді AW layer шукає еквівалентні послідовності, і як тільки він вибирає одну, вибрана послідовність виконується. Якщо виконання обхідного шляху або еквівалентної послідовності спричиняє іншу помилку, цикл продовжується, доки одна еквівалентна послідовність не спричинить жодної помилки або поки не буде більше еквівалентних послідовностей для спроб. В останньому випадку розробник має виправити помилку та підтримувати список еквівалентних послідовностей, після чого слід розгортати виправлений компонент і оновлювати список еквівалентних послідовностей, щоб зробити офлайнові зміни доступними для рівня AW у працюючій системі.

ВИСНОВКИ

Фактична підтримка самовідновлення протягом усього життєвого циклу самоадаптивних програмних систем вимагає реконцептуалізації способу їх розробки. Тому ми представили перше інтегроване рішення задачі, відповідні абстракції для автономних і он-лайн процесів, а також деталі основних етапів рішення цих задач щодо реконцептуалізації програмних процесів для самоадаптивних програмних систем. Крім того, для вирішення проблем, пов'язаних із розробкою самоадаптивних програмних систем, ми запропонували підхід, заснований на моделюванні процесів та розробці програмного забезпечення на основі цінностей. Важлива частина цього підходу – це переплетення самоадаптивної програмної системи та її програмного процесу [19].

В якості майбутньої роботи ми плануємо розробити реконцептуалізацію процесів програмного забезпечення для самоадаптивних систем, наприклад, шляхом дослідження впливу перспективи онлайн/офлайн на найсучасніші підходи, методи та техніки для проектування процесів програмного забезпечення та розробки самоадаптивних систем. Маючи більш глибокі знання про переосмислені процеси програмного забезпечення, ми можемо працювати над формалізацією мови моделювання, щоб повністю охопити процес і його систему. Формальна мова є необхідною умовою для автоматизованого аналізу, який відповідає теорії програмної інженерії на основі цінностей. Таким чином, ми повинні адаптувати цю теорію до специфіки самоадаптивних систем і процесів для таких систем. Наприклад, нам потрібно подумати про контрольні показники та метрики спеціального призначення для оцінки процесів і відповідних самоадаптивних систем, а також їх значення на основі витрат і вигод. Тоді можемо сказати, що даний процес кращий за інший у визначенні, проектуванні, реалізації та виконанні процесів онлайн і офлайн для системи.

Окрім цих початкових напрямків, дослідження повинні, зокрема, включати наступні елементи. Потрібно краще зрозуміти, як виявити функціональні та нефункціональні вимоги до самоадаптивних систем, особливо щодо того, як вони повинні бути пов'язані з діяльністю онлайн та офлайн та їх залежностями. Це може призвести до модельного рішення, розробки розгортання, адаптації та еволюції цих систем. Потрібно краще зрозуміти залежність між самовідновленням, самоадоптацією та еволюцією програмного забезпечення, оскільки перше не передбачає заміни останнього. Для цього потрібні чіткі визначення (само)адаптації та еволюції, а також того, як обидва можуть бути інтегровані в процес самоадаптивної системи. Наприклад, може виникнути потреба в розумінні того, як розвивати систему на основі її адаптації під час виконання. Досвід адаптацій, виконаних у минулому, може запропонувати корисні знання для еволюції системи.

Усі ці дослідницькі напрямки сприяють нашій кінцевій меті ефективного та ефективного проектування самоадаптивних систем із належною підтримкою процесу програмного забезпечення.

Підсумкову характеристику напрямкам подальших досліджень можна сформулювати наступним чином:

- інтеграція штучного інтелекту для прогнозування можливих відмов та автоматичного реагування на них;
- використання блокчейн-технологій для підвищення безпеки самовідновлювальних систем;
- оптимізація алгоритмів самовідновлення для підвищення швидкодії програмних систем;
- дослідження впливу різних архітектурних підходів на ефективність самовідновлення.

Таким чином, перспективні дослідження повинні використовувати систематичні підходи до розробки самоадаптивних систем, які мають передбачувані результати щодо ефективності.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Pesonen J. Software product roadmapping. School of Engineering Science. 2023. P. 10. DOI: 10.13140/RG.2.2.35599.87208.

2. Fitzgerald B., Klaas-Jan S. Continuous Software Engineering: A Roadmap andAgenda. The Journal of systems and software Vol. 123 2017. P. 176–189.

3. Web.Kittlaus, H. & Fricker, S. Software Product Management The ISPMA-CompliantStudy Guide and Handbook. Berlin, Heidelberg: Springer Berlin Heidelberg. 2017. pp. 119-181

4. Gabriel, R.P., Northrop, L., Schmidt, D.C., Sullivan, K.: Ultra-large-scale systems. In: OOPSLA 2006: Companion to the 21st ACM SIGPLAN Symposium on Objectoriented Programming Systems, Languages, and Applications, ACM, New York. 2006. P. 632–634.

5. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, Springer, Heidelberg. 2009. pp. 1–26.

6. Ruban I., Volk M., Filimonchuk T., Ivanisenko I., Risukhin M., Romanenkov Yu. The Method for Ensuring the Survivability of Distributed Computing in Heterogeneous Computer Systems. 5th International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T), Kharkiv, Ukraine, October 9-12, 2018. DOI: 10.1109/INFOCOMMST.2018.8632099

7. Ramanpreet K., Dušan G., Tomaž K. Artificial intelligence for cybersecurity: Literature review and future research directions, Information Fusion, Vol. 97. 2023. № 101804. ISSN 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2023.101804.

8. Волк М.О., Мамчич О.О. Оцінювання енергетичних витрат у процесі використання мобільних пристроїв для хмарних обчислень.

Сучасний стан наукових досліджень та технологій в промисловості. 2023. № 1 (23).– с.72-81. DOI: https://doi.org/10.30837/ITSSI.2023.23.072

9. Steinder M., Sethi A. A survey of fault localization techniques in computer networks. Science of Computer Programming Vol.53 (2) 2004. P. 165–194. DOI: https://doi.org/10.1016/j.scico.2004.01.010

10. Волк М.О., Гора М.В., Лабазов В. Г., Міщенко А.В., Барсуков А.І., Голець В.В. Журналізація стану програм для самовідновлення паралельних програмних систем. Системи управління, навігації та зв'язку, 2023, випуск 2(72), с. 80-87. DOI:https://doi.org/10.26906/SUNZ.2023.2.080

11. Isazadeh A, Izadkhah H, Elgedawy I., Source Code Modularization Theory and Techniques. Springer. 2017. З. 265. ISBN 978-3-319-63344-2.

12. Robidoux, R., Xu, H., Xing, L. and Zhou, M., "Automated modeling of dynamic reliability block diagrams using colored Petri nets", IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans. 2010. Vol. 40. No. 2. pp.337-351.

13. K. S. Trivedi and A. Bobbio, "DSN 2016 Tutorial: Reliability and Availability Modeling in Practice," 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), Toulouse, France, 2016, pp. 263-263, DOI: 10.1109/DSN-W.2016.51.

14. Yakovyna, V., Fedasyuk, D., Nytrebych, O., Parfenyuk, I. Matselyukh, V. Software reliability assessment using high-order Markov chains. International Journal of Engineering Science Invention, 2014. Vol. 3(7). P.1-6.

15. Волк М.О., Гора М.В. Моделі управління ресурсами для забезпечення функціональної стійкості процесу розподілених обчислень. Вісник Херсонського національного технічного університету. No 4(87), 2023 с. 244-251. DOI: https://doi.org/10.35546/kntu2078-4481.2023.4.28

16. Johny Antony P, Harsh Dev. Estimating Reliability of Software System using object-oriented metrics. International Journal of Computer Science Engineering and Information Technology Research. Vol. 3. No 2. 2013. pp. 283-294.

17. Волк М.О., Гора М.В. Модифікований метод самовідновлення розподіленого програмного забезпечення в гетерогенних комп'ютерних системах. Сучасний стан наукових досліджень та технологій в промисловості. 2024. №1 (27). С. 5-17. DOI: https://doi.org/10.30837/ITSSI.2024.27.005

18. G. Baumgarten, M. Rosinger, A. Todino and R. de Juan Marín, "SPEM 2.0 as process baseline Meta-Model for the development and optimization of complex embedded systems," 2015 IEEE International Symposium on Systems Engineering (ISSE), Rome, Italy, 2015, pp. 155-162, doi: 10.1109/SysEng.2015.7302749.

19. Волк М.О., Саранча С.М., Гора М.В., Ковтун Є.І., Лабазов В.Г., Полозов Д.М., Моделі ресурсів та програмних завдань для систем підтримки функціональної стійкості розподілених інформаційних систем. Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки. Том 35(74) №3,Ч.1. 2024. С. 42-47. DOI: https://doi.org/10.32782/2663-5941/2024.3.1/08