Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет навчально-науковий центр заочної форми навчання
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

АТЕСТАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи забезпечення функціональної стійкості розподілених програмних систем

(тема)

Виконав:
студент II курсу, групи СПзм-18-2
Маркін М.В.
(прізвище, ініціали)

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. Волк М.О.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2020 р.

Харківський національний університет радіоелектроніки

Факультет навчально-науковий центр заочної форми навчання

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___ р.

ЗАВДАННЯ

НА АТЕСТАЦІЙНУ РОБОТУ

студентові Маркіну Максиму Владиславовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Методи забезпечення функціональної стійкості розподілених програмних систем

затверджена наказом по університету від "30" березня 2020 р. № 43 Стз

2. Термін подання студентом роботи до екзаменаційної комісії 18 травня 2020 р.

3. Вхідні дані до роботи
Моделі та методи забезпечення функціональної стійкості.
Моделі розподілених програмних систем
Перелік можливих відмов в програмних системах
Методи забезпечення надійності та відмово стійкості програмних систем

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз предметної області
2. Моделі забезпечення функціональної стійкості розподілених програмних систем
3. Метод забезпечення функціональної стійкості програмних систем
4. Впровадження запропонованого методу забезпечення функціональної стійкості в електронній комерції

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) 11 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	Отримання завдання у керівника	30.03.2020	
2	Огляд літератури	05.04.2020	
3	Аналіз предметної області та	15.04.2020	
	постановка задач		
4.	Розробка моделей	20.04.2020	
5.	Впровадження моделей	31.04.2020	
6.	Практичне застосування технології	10.05.2020	
7.	Оформлення пояснювальної записки	18.05.2020	
	та підготовка демонстраційних матеріалів		

Дата видачі завдання 30 березня 2020 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Волк М.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка атестаційної роботи: 67 с., 9 рис., 2 табл.,1 дод., 30 джерел.

ПРОГРАММНА СИСТЕМА, ФУНКЦІОНАЛЬНА СТІЙКІСТЬ, НАДІЙНІСТЬ, РОЗПОДІЛЕНІ КОМПЬЮТЕРНІ СИСТЕМИ, МОДЕЛІ, МЕТОДИ

Метою атестаційної роботи є збільшення надійності та відмово стійкості розподілених програмних систем шляхом розробки моделі та методу забезпечення функціональної стійкості.

У ході виконання атестаційної роботи досліджено існуючі засоби забезпечення надійності та відмово стійкості розподілених програмних систем; розроблено модель забезпечення функціональної стійкості розподіленої програмної системи, метод забезпечення функціональної стійкості розподіленої програмної системи, макет програмних засобів підтримки функціональної стійкості розподіленої програмної системи для систем електронної комерції.

ABSTRACT

Master’s thesis:: 67 pages, 9 figures, 2 tables, 1 appendices, 30 sources.

SOFTWARE SYSTEM, FUNCTIONAL STABILITY, RELIABILITY, DISTRIBUTED COMPUTER SYSTEMS, MODELS, METHODS

The purpose of certification work is to increase the reliability and fault tolerance of distributed software systems by developing a model and method of ensuring functional stability.

In the course of attestation work the existing means of ensuring the reliability and failure of the distributed software systems were investigated; a model for ensuring the functional stability of a distributed software system, a method for ensuring the functional stability of a distributed software system, a model of software for maintaining the functional stability of a distributed software system for e-commerce systems have been developed.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

АСУ – автоматизована система управління

ІС – інформаційна система

ІТ – інформаційна технологія

КСМ – комп'ютерні системи та мережі

AOP – aspect-oriented programming

СМО – система масового обслуговування

CBA – Component Based Architecture

CBSN – Computer-based Systems and Networks

DARPA – Defense Advanced Research Projects Agency

IEEE – Institute of Electrical and Electronics Engineers

MDA – Model-driven Architecture

ROC – Recovery-Oriented Computing

SLA – Service Level Agreement

SOA – Service-oriented architecture

QoS – Quality of Service

USSD – Unstructured Supplementary Service Data

ВСТУП

Відносно до комп’ютерних систем та мереж (KCM, CBSN), забезпечення функціональної стійкості є одним із аспектів набору можливостей, що демонструються автономними обчислювальними системами, часто представленими терміном «живучість», який найчастіше включає самозахист, самонастроювання, самолікування та само оптимізацію. Основа автономних обчислень спирається на біологічні аналогії, щоб описати автономний обчислювач як комп’ютер, який самовідновлюється, таким же чином, як і біологічний організм[1][2].

Система програмного забезпечення для самолікування - це можливість виявити, діагностувати та виправити (або принаймні пом’якшити) збої в послугах, які вона надає. Для широкомасштабних систем може існувати багато різних типів несправностей, і їхня різна природа часто вимагає розрізнених, підібраних підходів для виявлення, не кажучи вже про їх виправлення.

Автономні системи поширюють вищезазначене поняття самолікування, включаючи можливості адаптації до змін зовнішнього середовища, наприклад, для підтримки його працездатності або наявності ресурсів.

У цій роботі представлена структура моделювання для розробки та впровадження самолікування, орієнтуючись на програмні аспекти системи. Архітектури моделей використовуються для класифікації відмов програмного забезпечення та уточнення їх диспозицій, крім моделей, що фіксують базову функціональність системи. Самолікування досягається шляхом перетворення базових моделей, а також моделей самовідновлення до платформних впровадження інструментально для виявлення та усунення несправностей з метою пом’якшення наслідків відмов програмного забезпечення та підтримання рівня адекватного функціонування системи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Забезпечення функціональної стійкості програмних систем

Обчислювальні системи характеризуються вимогами забезпечення працездатності (надійності) не тільки їх структури як сукупності апаратно-програмних ресурсів (в тому числі резервованих), але і обчислювального процесу, зокрема при необхідності підтримки його безперервності, в тому числі в умовах збоїв, відмов і зовнішніх деструктивних впливів випадкового або зловмисного характеру.

Особливість інформаційної комп’ютерної системи полягає в необхідності розгляду її не тільки як загальнотехнічного об’єкта з вимогами до структурної і параметричної надійності, але і як об’єкта, що реалізує інформаційні та обчислювальні процеси з вимогою функціональної надійності.

Під функціональною надійністю увазі здатність правильно і своєчасно виконувати необхідні функціональні завдання (функції) з прийнятним рівнем безпомилковості в реальних умовах експлуатації при взаємодії з зовнішніми об’єктами [1].

На рівень функціональної надійності (виконання функцій відповідно до якості, що визначаються користувачем) впливають не тільки - і не стільки - відмови і збої апаратури, а й, більшою мірою, надійність (стійкість) процесів обробки зберігання та передачі даних.

Надійність обчислювального процесу багато в чому залежить від дисциплін обслуговування запитів, виділення необхідних для цього ресурсів і їх відповідності виконуваних завдань, потік яких по відношенню до обчислювальної системи може в ряді випадків розглядатися як зовнішній. Це в певній мірі відрізняє обчислювальні, інформаційні та телекомунікаційні системи від інших технічних об’єктів.

Таким чином, виникає необхідність розгляду інформаційних і комунікаційних систем як систем масового обслуговування, відмови в обслуговуванні запитів в яких обумовлюються не тільки відмовами і збоями технічних засобів, але і перевантаженнями, недостатністю ресурсів, що може привести до втрати запитів або до затримок обслуговування понад допустимих меж. При оцінці функціональної надійності облік критичності затримок обслуговування особливо значущий для систем реального часу, в яких умовою відмови може бути невиконання запиту за заданий час.

Відмінності комп'ютерних систем від інших технічних систем обумовлюються також особливостями відмов програмних засобів, багато в чому визначають стійкість функціонування ВС.

Особливістю інформаційно-управляючих обчислювальних систем є насамперед жорсткі вимоги до надійності і відмово стійкості при їх роботі у важких умовах експлуатації (що призводять до зростання інтенсивності відмов). Керуючі інформаційно-обчислювальні системи, як правило, працюють в масштабі реального часу, що не допускає перерви управління і затримки обчислювального процесу понад гранично допустимих значень. Таким чином, обчислення в реальному часі в ряді випадків обмежують можливості повторних обчислень після збоїв і помилок, і тим більше можливості відновлення і реконфігурації системи.

Розробка програмного забезпечення переживає революцію. Ми досягли критичного періоду, коли команда з розвитку, діяльності та бізнесу не може функціонувати без допомоги комп'ютерів. Навряд чи є компанія - принаймні успішна - яка не покладається на програмне забезпечення, тобто якість, безпека та ефективність програмного забезпечення стають все більш важливими для ведення ефективного бізнесу.

Для організацій, які існують десятиліттями і довше, це означає, що вони не тільки повинні спрямовувати інвестиції на технологічні інновації, але й повинні модернізувати та підтримувати існуючі програмні системи. Це є викликом для IT-бюджетів, а також для набору розробників програмного

забезпечення. Тому найкращі програмні інженери працюють над рефакторингом старих систем замість сприйняття майбутньому самостійно керуючих автомобілів або міст, пов’язаних із цифровим зв’язком, які допомагають нам жити кращим життям. Але що робити, якщо обмеження роботи над старими системами сприяють новому інноваційному підходу для придушення цього тягаря для наступних поколінь?

З’являється програмне забезпечення для функціональної стійкості. Буквально десятиліття тому технічне співтовариство було невдоволено збуджене можливістю того, що програмна система може використовувати розроблену логіку не лише для виявлення власних збоїв, а й для автоматичного виправлення їх, не маючи втручання людини. Застосування програмного забезпечення для забезпечення функціональної стійкості у світі сьогодні створює технології для полегшення та спрощення модернізації програмного забезпечення і стає цікавим випадком використання програмних систем.

Концепція с функціональної стійкості з’явилася в інфраструктурному світі, щоб запобігти таким випадкам, як кібератаки або широкомасштабні відключення системи, які можуть спричинити порушення роботи клієнтів. Дані центри зазвичай оснащені інструкціями про відмову для автоматичного зберігання операцій у сервісі, а корпоративні мережі мають достатньо чутливий брандмауер для виявлення шкідливої діяльності, яка не дотримується звичайних протоколів. Контейнери та хмарні сервіси йдуть на крок далі, щоб уникнути виходу з ладу програм і служб, якщо, наприклад, різко збільшилася кількість транзакцій.

Ми вважаємо, що наступним кроком у забезпеченні функціональної стійкості є застосування його безпосередньо до вихідного коду для допомоги в розробці та вдосконаленні програмного забезпечення.

Розглянемо дослідницький проект IBM Israel SHADOWS, який опублікував результати ще в 2010 році. Ініціатива запропонувала усунути велику перешкоду модернізації та виправлення старого програмного

забезпечення, яке традиційно закликає інженерів витрачати години вручну, просіюючи старий код і діяти лише тоді, коли вони виявлять ситуації, в яких щось пішло не так. Ідея полягала у тому, щоб наслідувати поведінку людського організму та застосувати ці принципи до програмного забезпечення. Хоча проект працював «ззовні», не змінюючи вихідного коду, повного потенціалу вдалося досягти лише в тому випадку, якщо розробники все-таки знайшли час «забити» код інструкціями, щоб направити SHADOW для швидших і кращих діагнозів.

Робота команди, безумовно, була багатообіцяючою, але вона була занадто дорогою і занадто рано для ринку в той час, особливо коли офшоринг був на розпалі, а кількість робочих сил у таких країнах, як Індія, розширювалась на підтримку застарілого програмного забезпечення. Але через сім років ситуація змінилася. Витрати на власні повноваження не настільки ефективні, як раніше, і діловий світ змінився і став набагато гнучкішим. Компанії звикли обробляти швидкі зміни на всіх рівнях IT-системи - від переднього до заднього. Зараз якраз і є саме час, щоб переробити концепцію програмного забезпечення для забезпечення функціональної стійкості.

Незалежно від структури команди чи поточного процесу розвитку, починають шлях забезпечення функціональної стійкості, виконуючи основні три кроки, щоб зняти тиск на зусилля з обслуговування та дати розробникам більше свободи зосередитись на інноваціях.

1.1.1 Визначення можливих відмов на рівні вихідного коду

Перший крок у процесі забезпечення функціональної стійкості - виявлення моделей, схильних до створення небезпечних програм або внесення слабких місць у програму. Їх часто важко визначити на передній частині, оскільки вони розкриваються лише після того, як додаток з’являється у виробництві та під тиском користувачів. Громадські організації,

такі як CWE MITRE та Консорціум з якості IT-програмного забезпечення (CISQ), створили загальні мови та стандарти для вимірювання якості програмного забезпечення, які допомогають розпочати роботу. Встановлення базової лінії ефективності системи, стійкості, надійності та безпеки є основою створення програмного забезпечення для забезпечення функціональної стійкості.

Як тільки ці заходи будуть встановлені, команди можуть почати додавати систематичні засоби для аналізу тенденцій та виявлення загальних причин збоїв у застосуванні та збоїв у обслуговуванні. Це також прокладає шлях для машинного навчання і подальшої автоматизації аналізу шаблонів та прогнозування системних збоїв на основі зростаючих баз даних першопричин для поганого коду.

1.1.2 Принцип «розділяй і володарюй», зберігаючи повну інформацію

Команди, які вже почали впроваджувати мікросервіси (що передбачає розбиття програм на більш дрібні та легко мінливі компоненти), матимуть набагато простіший час впровадження процесу забезпечення функціональної стійкості. Однак у якийсь момент ці компоненти повинні бути складені разом для формування повноцінно функціонуючої програми.

Використання аналізу на рівні системи в якості гарантії якості перед тим, як повернути систему у виробництво, допоможе гарантувати, що нові дефекти програмного забезпечення або поганий код не повернуть систему до деградації. Іншим способом замислитись про аналіз системного рівня в цьому випадку може бути створення архітектурного плану перед впровадженням у виробництво.

1.1.3 Використовування забезпечення функціональної стійкості як плацдарма для майбутніх інновацій

Включення характеристик автоматизації забезпечення функціональної стійкості може допомогти створити основу для додаткових інновацій у організації. Наприклад, програмування програмного забезпечення для самостійного звітування про нерегулярну діяльність або логіку, яка не відповідає набору правил, можливо впровадити нові технології, такі як міграція хмари. Наразі акт міграції програмних програм у хмарне середовище передбачає окремий аналіз та перероблення коду, щоб переконатися, що програма буде працювати належним чином у хмарі. Уявіть, якби ці готові до хмари додатки могли самостійно проаналізувати та сказати команді розробників, що потрібно виправити перед міграцією. Це може заощадити незліченну кількість людей та запобігти зриву бізнесу протягом усього хмарного переходу.

1.2 Класифікація збоїв програмного забезпечення

Першим кроком до забезпечення функціональної стійкості є автоматичне розпізнавання програмних збоїв. Стандарт IEEE 1044 [3] документує вичерпний перелік категорій та класифікацій програмних аномалій. Таксономії програмних збоїв, помилок та несправностей широко обговорювалися для багатьох різних типів програмних систем. Загальноприйняті визначення наступні. Відмова програмного забезпечення стосується дефекту в системі. Помилка - це невідповідність між спостережуваною поведінкою системи та її заданою поведінкою. Поломка програмного забезпечення виникає, коли надана послуга відхиляється від правильного обслуговування, а відхилення поведінки системи від вимог користувача. Відмова або помилка програмного забезпечення не обов’язково може спричинити збій програмного забезпечення[4].

Нижче наведені основні категорії несправностей програмного забезпечення, про які йдеться в літературі для різних типів програмних систем [5][6]:

- синтаксичні помилки: несправності інтерфейсу та конфігураційних параметрів програмної системи;

- семантичні недоліки: непослідовна поведінка та неправильні результати;

- несправності сервісу: несправності qos, несправності, пов’язані з угодами про захист послуг (sla), і порушення в режимі реального часу;

- несправності в спілкуванні/взаємодії: тайм-аут та послуга недоступна;

- винятки: винятки, пов’язані з введенням/виводом, та винятки, пов’язані із безпекою.

Синтаксична помилка виникає, коли структури повідомлень або параметри запитувача не узгоджуються з структурами постачальника. Хоча компілятори здатні фіксувати такі типи несправностей в одній програмі, вони можуть статися в компонентному програмному забезпеченні та веб-службах, де запитувач і постачальник можуть будуватися незалежно один від одного. З іншого боку, семантичні недоліки представляли невідповідність між оригінальним дизайном та наміром програмістів. Семантичні вади є складнішими для діагностики і часто потребують спеціалізованих приладів для їх виявлення.

Помилка сервісу виникає, коли продуктивність послуги, що надається, відхиляється від необхідної продуктивності послуги, як зазначено в якості обслуговування (QoS) або угоді про рівень обслуговування (SLA). Ці типи несправностей призводять до погіршення працездатності системи. Несправності також виникають через зв’язок та взаємодію між компонентами, службами та підсистемами як в розподіленій, так і в централізованій системі. Відсутність та безвідповідальність провайдерів та терміни запитів на послуги є прикладами помилок спілкування та взаємодії.

Невиправлені винятки - ще одне джерело помилок програмного забезпечення, які часто призводять до ненормального припинення роботи програм. Рамки винятків, визначені на мовах, таких як Java та C++, забезпечують як вбудовані, так і визначені користувачем винятки. Однак вони не забезпечують механізму запобігання несподіваному припиненню заявки через необроблені винятки. Різні рівні зернистості у рамках винятку полегшується специфікацією обробки винятків для відновлення після конкретних та загальних помилок програмного забезпечення.

1.3 Виявлення помилок програмного забезпечення

Методи виявлення різних програмних несправностей різняться залежно від типів несправностей. Для синтаксичних несправностей структури повідомлень або параметрів перевіряються на вимоги до проектування, а недотримання вимог до проектування призведе до синтаксичних помилок. Несправності зв’язку та взаємодії можуть бути виявлені сторонніми мережними суб’єктами. Наприклад, обгортки можуть бути додані до виклику веб-сервісу для виявлення доступності та чутливості постачальників послуг. Для несправностей, пов’язаних із винятками, рамки винятків, визначені в мовах програмування, забезпечують єдиний підхід до виявлення несправностей. Наприклад, C ++ та Java надають конструкції «спробу» («try-catch») для зйомки як конкретних, так і загальних винятків програмування. Рамку можна також розширити, щоб охопити визначені користувачем винятки.

Несправності сервісу вимагають моніторингу служб, щоб визначити, чи надані послуги відповідають необхідній угоді QoS або SLA. Для різних типів несправностей служби можуть знадобитися різні механізми виявлення. Наприклад, якщо виклик послуги повинен бути завершений протягом певного часового періоду, виклик може бути приурочений таким чином, що виявляється невиконання відповіді протягом необхідного періоду часу.

Виявлення смислових недоліків складніше. Для розуміння правильної та неправильної поведінки компонента, послуги чи системи потрібні спеціалізовані інструментальні та/або доменні знання. Для виявлення несправностей також можна застосувати різні загальні підходи. Одним із поширених підходів є виконання інструментарії вихідного коду. Додатковий код вставляється у вихідний код для полегшення виявлення програмних аномалій. Наприклад, програмування, орієнтоване на аспекти (AOP) [7], може застосовуватися для приєднання додаткових функцій до системи з метою виявлення продуктивності та несправностей системи. Для виявлення несправностей програмного забезпечення також можуть застосовуватися датчики та збір та обробка подій. Більшість голосів [8], наприклад, можна знайти в телекомунікаційних системах для виявлення програмного забезпечення або системної аномалії для забезпечення високої надійності та доступності.

Невдача прогнозування може також використовуватися для посилення вищезазначених методів виявлення. Це включає як навчання, так і міркування про невдачі. Перед тим, як виникнуть помилки, можна вжити попереджувальних та коректівних дій, щоб уникнути втрати функціональних можливостей або порушення послуг [5].

Якщо припускати, що в програмному забезпеченні якісь помилки все ж будуть, то найкраща (після попередження помилок) стратегія - включити засоби виявлення помилок в саме програмне забезпечення.

Більшість методів направлено по можливості на негайне виявлення збоїв. Негайне виявлення має дві переваги: можна мінімізувати вплив помилки і наступні труднощі для людини, якій доведеться витягувати інформацію про неї, знаходити її і виправляти.

Заходи з виявлення помилок можна розбити на дві підгрупи: пасивні спроби виявити симптоми помилки в процесі «звичайної» роботи програмного забезпечення та активні спроби програмної системи періодично обстежувати свої статки в пошуках ознак помилок.

Пасивне виявлення. Заходи з виявлення помилок можуть бути прийняті на декількох структурних рівнях програмної системи. Тут ми будемо розглядати рівень підсистем, або компонентів, тобто нас будуть цікавити заходи по виявленню симптомів помилок, що вживаються при переході від одного компонента до іншого, а також всередині компонента. Все це, звичайно, можна застосувати також до окремих модулів всередині компонента.

Розробляючи ці заходи, ми будемо спиратися на наступне:

- взаємна недовіра. Кожен з компонентів повинен припускати, що всі інші містять помилки. Коли він отримує якісь дані від іншого компонента або з джерела поза системи, він повинен припускати, що дані можуть бути неправильними, і намагатися знайти в них помилки;

- негайне виявлення. Помилки необхідно виявити якомога раніше. Це не тільки обмежує наноситься ними шкоду, але і значно спрощує завдання налагодження;

- надмірність. Всі засоби виявлення помилок засновані на деякій формі надмірності (явній або неявній).

Коли розробляються заходи по виявленню помилок, важливо прийняти узгоджену стратегію для всієї системи. Дії, що вживаються після виявлення помилки в програмному забезпеченні, повинні бути однаковими для всіх компонентів системи. Це ставить питання про тому, які саме дії слід зробити, коли помилку виявлено. Найкраще рішення - негайно завершити виконання програми або (в разі операційної системи) перевести центральний процесор в стан очікування. З точки зору надання людині, налагоджують програму, наприклад системному програмісту, найсприятливіших умов для діагностики помилок негайне завершення представляється найкращою стратегією. Звичайно, в багатьох системах подібна стратегія буває недоцільною (наприклад, може виявитися, що припиняти роботу системи не можна). В такому випадку використовується метод реєстрації помилок. Опис симптомів помилки і «моментальний знімок» стану системи зберігаються в

зовнішньому файлі, після чого система може продовжувати роботу. Цей файл пізніше буде вивчений обслуговуючим персоналом.

Завжди, коли це можливо, краще призупинити виконання програми, ніж реєструвати помилки (або забезпечити як додаткову можливість роботу системи в будь-якому з цих режимів). Різниця між цими методами проілюструємо на способах виявлення причин виникає іноді скреготу вашого автомобіля. Якщо автомеханік знаходиться на задньому сидінні, то він може обстежити стан машини в той момент, коли скрегіт виникає. Якщо ви вибираєте метод реєстрації помилок, завдання діагностики стане складніше.

Активне виявлення помилок. Не всі помилки можна виявити пасивними методами, оскільки ці методи виявляють помилку лише тоді, коли її симптоми піддаються відповідної перевірці. Можна робити і додаткові перевірки, якщо спроектувати спеціальні програмні засоби для активного пошуку ознак помилок в системі. Такі кошти називаються засобами активного виявлення помилок.

Активні засоби виявлення помилок зазвичай об’єднуються в діагностичний монітор: паралельний процес, який періодично аналізує стан системи, з метою виявлення помилки. Великі програмні системи, що управляють ресурсами, часто містять помилки, призводять до втрати ресурсів на тривалий час. Наприклад, управління пам’яттю операційної системи здає блоки пам’яті «в оренду» програмами користувачів і іншим частинам операційної системи. Помилка в цих самих «інших частинах» системи може іноді вести до неправильної роботі блоку управління пам’яттю, що займається поверненням зданої раніше в оренду пам’яті, що викликає повільне виродження системи.

Діагностичний монітор можна реалізувати як періодично виконувану задачу (наприклад, вона планується на кожну годину) або як завдання з низьким пріоритетом, яка планується для виконання в той час, коли система переходить в стан очікування. Як і раніше, виконуються монітором конкретні перевірки залежать від специфіки системи, але деякі ідеї будуть зрозумілі із

прикладів. Монітор може обстежити основну пам’ять, щоб виявити блоки пам’яті, що не виділені жодної з виконуваних завдань і не включені в системний список вільної пам’яті. Він може перевіряти також незвичайні ситуації: наприклад, процес не планувався для виконання протягом деякого розумного інтервалу часу. монітор може здійснювати пошук «загубилися» всередині системи повідомлень або операцій введення-виведення, які незвично довгий час залишаються незавершеними, ділянок пам’яті на диску, які не позначене як виділені і не включені в список вільної пам’яті, а також різного роду дивацтв в файлах даних.

Іноді бажано, щоб в надзвичайних обставинах монітор виконував діагностичні тести системи. Він може викликати певні системні функції, порівнюючи їх результат з заздалегідь визначеним і перевіряючи, наскільки розумно час виконання. Монітор може також періодично пред’являти системі «порожні» або «легкі» завдання, щоб переконатися, що система функціонує хоча б найпримітивнішим чином.

1.4 Відновлення функціонування програмної системи

Існують різні підходи до усунення несправностей із програмним забезпеченням: запобігання несправностей, усунення несправностей та відмовостійкість. Попередження та усунення несправностей мають на меті досягти безвідмовного програмного забезпечення шляхом надійної конструкції та ретельного тестування. Метою відмовостійкості є забезпечення постійної роботи системи за наявності несправностей.

Досягати при забезпеченні функціональної стійкості при виникненні несправності програмного забезпечення система повинна мати можливість повернутися до одного зі своїх нормальних робочих станів. Це вимагає, щоб система знала, що таке нормальні робочі стани та які необхідні дії потрібно виконати для повернення до нормальних робочих станів. Відновлення помилок, вперед або назад, можна використовувати для того, щоб система

могла дійти стану без помилок. Надлишок може також використовуватися для компенсації помилок. Для врятування стану процесу чи компонента, що підлягає відновленню, коли виникають помилки, як це часто потрібно при обробці транзакцій, можна використовувати контрольну точку.

Існують різні рівні деталізації для усунення несправностей. Наприклад, обробку винятків можна оголосити для всіх типів винятків, від найбільш специфічних до найзагальніших. Для окремого винятку можуть бути вжиті конкретні дії для ефективного відновлення після виключення. Однак, для загальних винятків, дії з відновлення можуть бути менш ефективними, менш точними та більш дорогими, як, наприклад, скидання або перезапуск складної підсистеми.

Розглядаючи проблему забезпечення функціональної стійкості розподілених програмних систем, зокрема інформаційних систем (ІС) в першу чергу треба розібратися з поняттями відмовостійкості та надійності.

Надійність - властивість системи зберігати в часі у встановлених обмеженнях здатність виконувати свої функції в заданих режимах і умовах використання, технічного обслуговування, експлуатації, зберігання і розміщення.

Питання надійності комп’ютерних систем існує вже кілька десятиліть, і воно особливо загострилася з впровадженням складних систем масового обслуговування (СМО). Побудова і експлуатація такої техніки без спеціальних заходів щодо забезпечення її надійності не використовується. Небезпека є не тільки в тому, що інноваційна техніка не буде підтримувати свої функції, але й головним чином в тому, що відмови в її роботі, в тому числі і некоректна робота, може привести до катастрофічних наслідків. З урахуванням цього при розробці, створенні та використанні таких систем повинні вживатися відповідні заходи щодо забезпечення підвищення надійності цих систем.

Забезпечення надійності технічних компонентів комп’ютерних систем реалізується апаратним і програмним способами. У першому випадку

система використовує зазвичай апаратну надмірність: всі операції виконуються паралельно на однорідних компонентах системи, а результати роботи окремих компонентів потім порівнюються, що дозволяє виявити наявні помилки; в разі виходу з ладу будь-якого компонента системи його резервні аналоги зазвичай продовжують роботу без зупинки, а відмовивший компонент системи замінюється на працездатний. Програмний засіб передбачає: постійний моніторинг функцій системи, послідовне в часі виконання одних самих обчислювальних процесів і дублювання даних; автоматичне відновлення компонентів операційних систем, додатків та збережених даних.

Для забезпечення надійності таких засобів найчастіше виконується: резервування (дублювання) комп'ютерних засобів; використання стандартних протоколів роботи та відновлення ІС; застосування спеціалізованих технічних засобів захисту даних. Для забезпечення надійності функціонування комп'ютерної системи потрібні ретельне стеження та контроль. Наприклад, для комп'ютерів в якості найбільш ефективних заходів комплексного забезпечення надійності програмних систем можна назвати кластеризацию комп'ютерів та мереж і використання стійких до відмов апаратних засобів.

У теорії надійності вельми важливу роль відіграє розподіл елементів і систем на відновлювані і невідновлювані. Вони дозволяють обґрунтовано вирішувати завдання забезпечення надійності.

При аналітичному методі оцінки, основними показниками надійності є: ймовірність безвідмовної роботи і середній наробіток на відмову, які визначаються за відомими параметрами потоків відмов елементів, що входять в дану комп'ютерну систему. Однак для автоматизованих систем управління (АСУ), інформаційних мереж і обчислювальної техніки, цих понять для характеристики надійності недостатньо. У практиці створення використання АСУ застосовуються додаткові поняття, з урахуванням яких можна повною мірою представити комплексне поняття «надійність».

Розглянемо ці поняття:

1. Живучість - властивість об’єкта зберігати працездатність (повністю або частково) в умовах несприятливих впливів, не передбачених нормальними умовами експлуатації. Головний сенс вимоги до живучості об’єкту полягає не тільки в тому, щоб він тривалий час безперервно без відмови працював в нормальних умовах експлуатації і, щоб його можна було швидко відремонтувати, але також і в тому, щоб він в ненормальних умовах експлуатації зберігав працездатність, хоча б і обмежену.

2. Достовірність інформації, яка видається об’єктом. При роботі обчислювальної машини або тракту передачі інформації можуть бути відсутні відмови. Тому об’єкт може мати високу безотказностью, хорошою довговічністю і ремонтопридатністю. Однак в ньому можуть мати місце збої, які спотворюють інформацію. «Псуватися» не апаратура, а інформація. Це не менш небезпечна «поломка».

При дослідженні надійності часто ставиться завдання визначити причини, що призводять до врахування тієї чи іншої сторони надійності. Без цього неможливо наладити коректну програму робіт з підвищення надійності. Це призводить до поділу надійності на:

- апаратну надійність, обумовлену станом апаратури;
- програмну надійність об’єкта, обумовлену. станом програм;
- надійність об’єкта, обумовлену якістю обслуговування.

Відмовостійкість - властивість комп’ютерної системи зберігати свою працездатність після відмови одного або декількох складових компонентів. Відмовостійкість визначається кількістю будь-яких послідовних одиничних відмов компонентів, після якого зберігається працездатність системи в цілому. Основний рівень відмово стійкості забезпечує захист від відмови одного будь-якого елементу - виняток єдиної відмови одного з елементів системи. Базовий спосіб підвищення відмовостійкості - надмірність. Найбільш відомий ефективний метод надмірності - апаратна надмірність, яка досягається шляхом апаратного резервування.

Відмовостійкість слід відрізняти від відмовобезпечності - здатності системи при відмові деяких наявних частин переходити в режим роботи, що не уявляє небезпеки для людей, навколишнього середовища або матеріальних цінностей. Однак в реальних системах ці дві вимоги можуть іноді виступати спільно.

Відмовостійкість пов’язана з наступними технічними характеристиками систем:

- коефіцієнт готовності, який показує, яку частку часу від загального часу служби система знаходиться в робочому стані;

- надійність системи, яка визначається, наприклад, як ймовірність відмови в одиницю часу.

Відмовостійка архітектура з точки зору інженерії забезпечується методами проектування відмовостійких систем, які здатні продовжувати виконання запланованих операцій (можливо, з пониженням ефективності) при відмові їх компонентів [1]. Термін часто використовується для опису комп’ютерних систем, спроектованих продовжувати роботу в тій чи іншій мірі, з можливим зменшенням пропускної спроможності або збільшенням часу відгуку, в разі відмови частини системи. Це означає, що система в цілому не припинить своє функціонування при виникненні збоїв з апаратною або програмною частиною. Приклад з іншої області: структура, здатна зберігати свою цілісність при пошкодженні від корозії або втоми матеріалу, наявності виробничих дефектів або зіткнення з яким-небудь об’єктом.

Надмірністю або резервуванням називають функціональність, в якій немає необхідності при безвідмовної роботи системи.

Прикладами можуть бути запасні частини, які автоматично вмикаються в роботу, якщо основна відмовляє. Зокрема, великі вантажівки можуть втратити шину без серйозних наслідків. На них встановлено багато шин, і втрата однієї не є критичною (за винятком передньої пари, яка служить для поворотів). Вперше ідея включення надлишкових частин для збільшення надійності системи була висловлена Джоном фон Нейманом в 1950-х роках.

Існує два типи надмірності: просторова й тимчасова. Надмірність простору зазвичай реалізується шляхом введення додаткових компонентів, даних або функцій, які не потрібні при безвідмовному функціонуванні. Додаткові (надлишкові) компоненти можуть бути апаратними, програмними та у вигляді даних. Тимчасова надмірність реалізується за допомогою повторних обчислень або пересилки даних, після чого результат порівнюється з копією попереднього або паралельного обчислення.

З точки зору реакції на відмову надлишкові системи можна розділити на системи з пасивним резервуванням і системи з активним резервуванням. Для системи з пасивним резервуванням характерно усунення впливу відмов блоків без зміни структурної, функціональної організації. Слід підкреслити, що в сучасних відмовостійких системах методи пасивного резервування в силу своєї неефективності в чистому вигляді практично не використовуються. Обчислювальна система з активним резервуванням характеризуються виконанням певних дій зі зміни характеру її функціонування, функціональної та структурної конфігурації системи для усунення наслідків виникли відмов.

Суть методів активного резервування полягає в управлінні структурним резервом, тобто в структурній реконфігурації обчислювальної системи. До цих методів належать дублювання, гібридне резервування і т.д.. Питанням управління структурним резервом в процесі функціонування обчислювальної системи присвячені численні дослідження. При такому підході підсистема контролю функціонування вирішує питання про виключення відмовили блоків з процесу функціонування, підключення замість них працездатних блоків з відновленням стану, що передував відмови, і відновлення процесу функціонування обчислювальної системи.

Проектування кожного компонента як відмовостійкого приносить в систему деякі недоліки: збільшення ваги, вартості, енергоспоживання, ціни і часу, витраченого на проектування, перевірку і випробування. Для визначення того, які компоненти слід проектувати відмовостійкості, існує

набір тестів:

- наскільки важливий компонент? Наприклад, в автомобілі радіо не є важливою частиною, тому воно не потребує відмовостійкості.

- наскільки велика ймовірність відмови компонента? Деякі частини, такі як карданний вал автомобіля, ламаються вкрай рідко, отже, не потребують відмовостійкості.

- наскільки дорогим буде забезпечення відмовостійкості компонента? Наприклад, забезпечення відмовостійкості двигуна автомобіля занадто дорого і економічно, і конструктивно (вага, розмір).

Прикладом елемента системи, що пройшов за всіма пунктами, можуть служити ремені безпеки автомобіля. Ми не замислюємося про первинну системі, що утримує пасажирів - гравітації, але якщо машина перевертається або відчуває серйозні перевантаження, то первинний метод утримання може відмовити. Вторинна система, що утримує пасажирів на місцях під час таких подій, важлива для їх безпеки - таким чином, перший пункт підтверджується. Випадки викиду пасажирів були звичайною справою до введення ременів безпеки - підтверджується другий пункт. Ціна установки додаткових заходів утримання, таких як ремені безпеки, невелика і в плані витрат, і в плані технологічності - підтверджується третій пункт. У підсумку робиться висновок, що установка ременів безпеки на всі засоби пересування - відмінна ідея.

Іноді забезпечення відмовостійкості апаратури вимагає, щоб вийшли з ладу частини були вилучені і замінені новими, в той час як система продовжує працювати (в області комп'ютерів відомо як гаряча заміна). Це реалізується за допомогою однієї надлишкової частини і відома як стійка в одній точці, велика частина відмовостійких систем виглядають саме так. У таких системах середній час між поломками має бути досить велике, щоб оператори могли встигнути виконати ремонт до того, як резервна частина теж вийде з ладу. Така методика допомагає, якщо час між поломками максимально можливе, що не є необхідною умовою для побудови

відмовостійкості системи.

Переваги відмовостійких рішень очевидні, але також у них існують і недоліки.

Перешкоди у виявленні однокомпонентних неполадок. Наприклад, водій автомобіля може не помітити, що шина проколота, якщо використовується будь-яка відмовостійкості система. Проблема може бути вирішена шляхом додавання спеціальної системи для виявлення неполадок. У разі шини, система стежить за тиском в камерах і попереджає водія, якщо воно падає. Альтернативою може служити ручна система виявлення неполадок, наприклад огляд водієм шин на кожній зупинці.

Перешкоди у виявленні багатокомпонентних неполадок. Відмовостійкість одного компонента може заважати виявленню неполадок в іншому. Наприклад, якщо частина В виконує якусь операцію на основі даних з частини А, то відмовостійкість частини В може приховати проблему, що виникла в А. Якщо в подальшому частину В буде замінена на менш отказоустойчивую, то система може раптово відмовити, при цьому буде здаватися, що проблема полягає в новій частині В. І тільки після ретельної перевірки системи стане ясно, що проблема була в частині А.

Зменшення пріоритету виправлення неполадки. Навіть якщо оператор знає про наявність проблеми, існування відмовостійкості системи зменшить важливість виправлення проблеми. А якщо проблеми не виправляти, то це, скоріше за все, призведе до повної відмови системи, коли відмовостійкості частина зламається повністю або ж коли зламаються всі надлишкові частини.

Складність перевірки. Для деяких вкрай важливих відмовостійких систем, таких як ядерний реактор, немає простого шляху, щоб упевнитися, що запасні частини знаходяться в робочому стані. Сумнозвісним прикладом служить Чорнобильська аварія, коли оператори перевіряли аварійну систему охолодження шляхом відключення основної та допоміжної систем. Аварійна система не витримала, що вилилося в перегрів ядра і великий викид радіації.

Ціна. І відмовостійкості, і надлишкові компоненти збільшують вартість

системи. Це може бути як ціна, так і інші фактори, такі як вага. Наприклад, пілотовані космічні кораблі мають так багато надлишкових та відмовостійкості частин, що їх вага сильно збільшується в порівнянні з безпілотними апаратами, які не вимагають такого рівня безпеки.

Неякісні компоненти. Відмовостійка архітектура може дозволити використання неякісних частин, які в іншому випадку зробили б систему непрацюючої. Хоча ця практика може використовуватися для пом’якшення збільшення витрат, використання декількох неякісних частин може знизити надійність системи до вихідного рівня або навіть нижче.

Наступний крок - методи виправлення помилок. Після того як помилка виявлена, або вона сама, або її наслідки повинні бути виправлені програмним забезпеченням. виправлення помилок самою системою - плідний метод проектування надійних систем апаратного забезпечення. Деякі пристрої здатні виявити несправні компоненти і перейти до використання ідентичних запасних. Аналогічні методи незастосовні до програмного забезпечення внаслідок глибоких внутрішніх відмінностей між збоями апаратури і помилками в програмах. Якщо деякий програмний модуль містить помилку, ідентичні «запасні» модулі також будуть містити ту ж помилку.

Інший підхід до виправлення пов’язаний зі спробами відновити руйнування, викликані помилками, наприклад спотворення записів в базі даних або керуючих таблицях системи. користь від методів боротьби з спотвореннями обмежена, оскільки передбачається, що розробник заздалегідь вгадає кілька можливих типів спотворень і передбачить програмно реалізовані функції для їх усунення. Це схоже на парадокс, оскільки, якщо знати заздалегідь які помилки виникнуть, можна було б вжити додаткових заходів щодо їх попередження. якщо методи ліквідації наслідків збоїв не можуть бути узагальнені для роботи з багатьма типами спотворень, найкраще направляти сили і засоби на попередження помилок. Замість того, щоб, розробляючи операційну систему, оснащувати її засобами виявлення і відновлення ланцюжка перекручених таблиць або керуючих

блоків, слід було б краще спроектувати систему так, щоб тільки один модуль мав доступ до цієї ланцюжку, а потім наполегливо намагатися переконатися в правильності цього модуля.

Основне допущення програмування, стійкого до програмних помилок, полягає в тому, що як би добре не була спроектована і реалізована програма, в ній обов’язково буде мати декілька залишкових помилок. А раз так, то модулі програми, які можуть дати збій, повинні мати «резервний запас». З цією метою модуль проектується у вигляді блоків відновлення. Кожен блок відновлення містить пропускної тест і один або кілька варіантів реалізації. Основний варіант ініціюється при виклику блоку відновлення, і коли його виконання завершується, відбувається перевірка значення пропускного тесту. Якщо він дає «істину», то вважається, що виконання блоку відновлення успішно завершено. Якщо ж тест дає «брехня», то ініціюється інший варіант, за яким слід визначення значення пропускного тесту і т.д., і так до успішного виконання блоку відновлення. Якщо ж жоден варіант не пройшов пропускного тесту, то блок відновлення розглядається як помилковий і починається виконання іншого варіанту викликається модуля. Застосовується також і інший технічний прийом: написані, часто незалежно, кілька різних сегментів програми, кожен з яких призначений для виконання однієї функції. Програма будується з цих сегментів. Перший сегмент, званий первинним, виконується першим. За ним слідує приймальне випробування результату обчислень першого сегмента. Якщо випробування пройшло успішно, тоді результат приймається і передається до подальшим частинам системи. Якщо випробування було невдалим, будь-які побічні ефекти першого сегмента скидаються, і виконується другий сегмент, званий перший альтернативний. За ним також слід приймальне випробування, результати якого розглядаються як в першому випадку. Якщо необхідно, можуть бути реалізовані інші альтернативні прийоми.

1.5 Використання функціональної стійкості програмного забезпечення в електронній комерції

Більшість компаній шукають спільних цілей, таких як розширення ринків, поліпшення обслуговування клієнтів, зниження витрат і підвищення продуктивності. Тому, щоб досягти цих концепцій, вони вирішили піти на «електронну комерцію» або бізнес, що ведеться в режимі он-лайн та впроваджувати властивості цієї системи у своїх системах. Банки вживають заходів для розширення використання технологій мереж у своїх бізнес-операціях, як і інші компанії. Однак банки починають використовувати електричну технологію для доставки традиційних банківських продуктів більш ефективними способами. Крім того, деякі банки зробили подальший крок у розробці нових продуктів, розроблених спеціально для полегшення участі в електронній комерції своїх клієнтів. Завдяки цім причинам будь-які банки чи фінансові компанії мають бажання торгувати через електронний банкінг. З іншого боку, незважаючи на те, що електронний банкінг стає важливим аспектом світової торгівлі, у деяких країнах він недостатньо прийнятний, особливо в країнах, що розвиваються. У цих країнах майбутнє цієї концепції виглядає світлим, але торговці та клієнти стикаються з деякими бар’єрами для її обґрунтування, такими як менш надійна телекомунікаційна інфраструктура та джерела живлення, менший доступ до механізмів оплати в Інтернеті та порівняно високі витрати на персональні комп’ютери та доступ до Інтернету для деяких банківські послуги.

Повага до інтенсивної конкуренції у банківській галузі та їх зусилля щодо утримання існуючих клієнтів, а також залучення нових клієнтів змушують їх розвивати електронний банк у комплексній стратегії інфраструктури послуг[1]. Існує багато визначень для електронного банкінгу, наприклад «електронний банкінг» - це надання банками інформації та послуг банками клієнтам через різні платформи доставки, які можна використовувати з різними термінальними пристроями, такими як

персональний комп’ютер та мобільний телефон із браузером або настільне програмне забезпечення, телефонне або цифрове телебачення[2].

У банківському плані електронний банкінг дозволяє клієнтам виконувати безліч простих банківських функцій в будь-який час і в будь-якому місці. Незважаючи на те, що банки можуть побажати своїм клієнтам перенести більшість своїх транзакцій на дешевіші канали електронного банкінгу, клієнти часто змішують використання декількох каналів таким чином, що не обов’язково відповідає тому, що можуть захотіти банки. З точки зору клієнтів важливо, щоб банківські операції були доступними цілодобово, 365 днів на рік, незважаючи на заплановані або позапланові відключення. Багато банків мають географічно розподілені центри обробки даних, які потребують інформації по центрах. Щоб досягти цього, банківські платформи повинні бути достатньо надійними, щоб забезпечити «п’ять дев’яток» наявності або система повинна бути доступною протягом 99,999% часу на рік.

Кілька попередніх досліджень показують, що доступність нових каналів обслуговування або інша сервісна інфраструктура змінюють поведінку споживачів. Наприклад, [3] показало, що прийняття споживачами електронного банкінгу мотивує їх відкривати більше рахунків та збільшувати їх загальний попит на банківські послуги, що дозволяє припустити, що між різними каналами існує певна стратегічна синергія, яку банки можуть вивчити далі для отримання конкурентоспроможних послуг. перевага [1].

В результаті система електронного банкінгу вирішує декілька тенденцій, що розвиваються: попит клієнтів на будь-який час та будь-яку послугу, імперативи часу на ринок продуктів та все більш складні завдання інтеграції бек-офісів. Деякі властивості мають особливий ступінь важливості в електронному банкінгу: безпека, конфіденційність, зручність, дизайн, вміст, швидкість, збори та збори, доступність, надійність, QoS 1 тощо. Крім того, є деякі фактори, які викликають задоволення клієнтів, такі як задоволення, вірність, WoM2. У цій роботі ми зупинимось на двох згаданих

параметрах впровадження електронного банкінгу, які є доступністю та якістю послуг.

Доступність визначається як здатність користувачів отримувати доступ до інформації та послуг з будь-якого терміналу, що залежить від багатьох факторів. До них належить формат контенту; апаратне, програмне забезпечення та налаштування користувача; достатні та впевнені зв’язки; екологічні умови та здібності та обмеження користувача [2, 4].

Існує кілька моделей якості для систем, одна з найбільш релевантних була встановлена в [5], класифікуючи якість на структурованому наборі характеристик і під характеристик наступним чином:

- функціональність: придатність, точність, інтероперабельність, відповідність, безпека;
- надійність: зрілість, відновлюваність, помилка толерантність;
- корисність: навчання, розуміння, працездатність;
- ефективність: поведінка часу, поведінка ресурсів;
- технічне обслуговування: стабільність, аналіз, змінність, заповітність;
- переносність: встановлення, заміна, адаптивність, відповідність.

Якщо система дотримується умов якості обслуговування, то вона може досягти максимального задоволення клієнтів. Тому основним мотивом є те, що задоволені клієнти приносять більший прибуток. Це виглядає в ланцюговій реакції задоволеності клієнтів, яка показана на рисунку 1.1.

Забезпечення функціональної стійкості - це фраза, застосована до процесу одужання, мотивована та спрямована клієнтом, керована часто лише інстинктом. Самолікування - це механізм, який виявляє та усуває проблеми функціональності та послуг, пропоновані електронними системами. У деяких системах, що базуються на окремих компонентах і послугах, тестувати та аналізувати їх легко, на відміну від складних систем, які страждають від несподіваних проблем інтеграції, можуть призвести до тонких проблем з часом роботи, які важко запобігти та відновити.

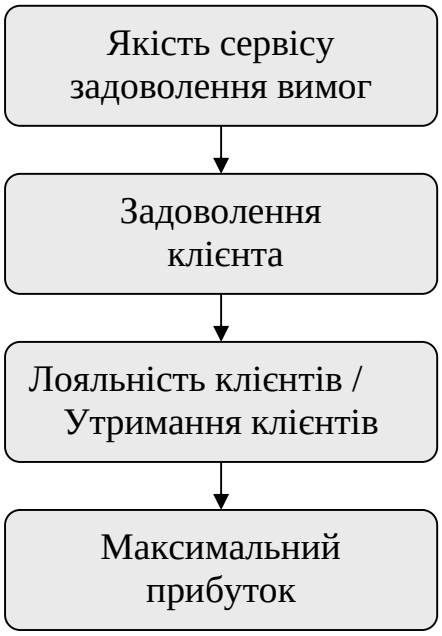

Рисунок 1.1 – Ланцюгова реакція задоволеності клієнтів

Механізми забезпечення функціональної стійкості зосереджені на виявленні та подоланні таких проблем. Внаслідок цього забезпечення функціональної стійкості перетворилося на властивість IT-систем, що дозволяє таким системам діагностувати себе та реагувати на несправності, що забезпечує відновлення задовільного режиму роботи після виникнення несправності чи відмови. Таким чином, ми використовуємо нижній цикл для відновлення відмов з мінімальною залежністю від експерта. Рисунок 1.2 показує цикл забезпечення функціональної стійкості.

Багато компаній підтримує електронну комерцію або бізнес, що ведеться в режимі он-лайн через електронні мережі, як засіб розширення ринків, поліпшення обслуговування клієнтів, зниження витрат та підвищення продуктивності. Як і інші компанії, банки вживають заходів для розширення використання технологій інформатики у своїх бізнес-операціях. З іншого боку, в останні роки прийняття електронного банкінгу почало відбуватися досить широко, як канал розподілу фінансових послуг завдяки швидкому прогресу в IT та інтенсивному конкурентному банківському ринку. Для

банку, який хоче залишатися в конкурентній боротьбі, важливіше, щоб кожна послуга електронного банкінгу мала деякі особливості, такі як надійність, доступність, розширюваність.

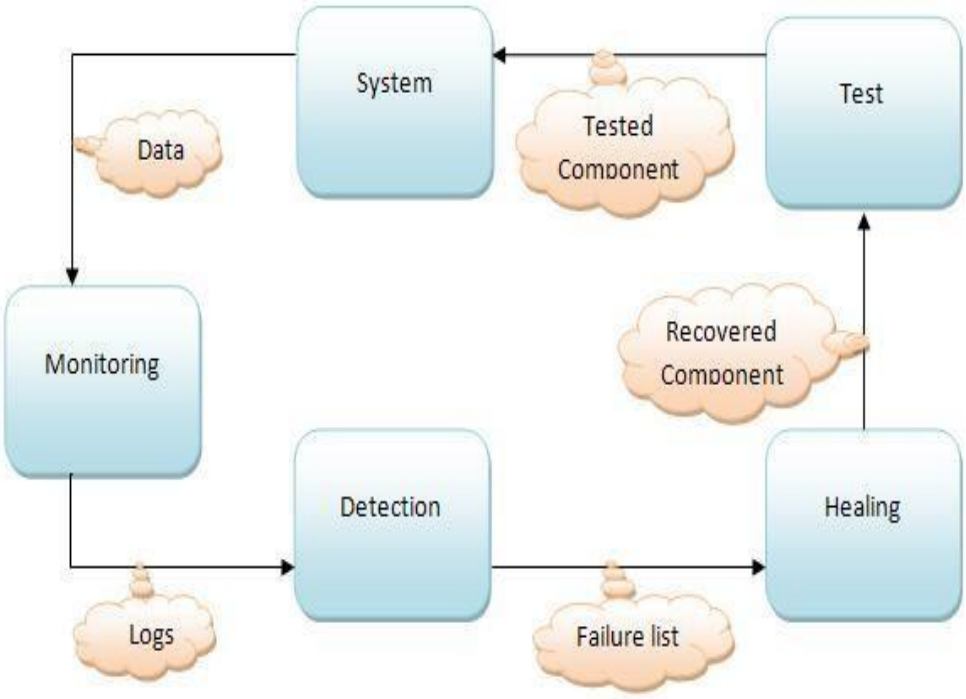

Рисунок 1.2 – Цикл забезпечення функціональної стійкості

Отже, в роботі представлено нову стійку систему систему електронного банкінгу, яка використовувала концепцію забезпечення функціональної стійкості в умовах обмежань ефективними способами. Крім того, деякі банки зробили подальший крок у розробці нових продуктів, розроблених спеціально для полегшення участі в електронній комерції своїх клієнтів.

1.6 Мета та задачі атестаційної роботи

Зважаючи на попередній аналіз предметної області, сформульовано мету та задач магістерської атестаційної роботи.

Метою роботи є збільшення надійності та відмовостійкості

розподілених програмних систем шляхом розробки моделі та методу забезпечення функціональної стійкості.

Згідно встановленої мети необхідно вирішити наступні задачі:

- дослідити існуючі засоби забезпечення надійності та відмово стійкості розподілених програмних систем;
- розробити модель забезпечення функціональної стійкості розподіленої програмної системи;
- розробити метод забезпечення функціональної стійкості розподіленої програмної системи;
- розробити макет програмних засобів підтримки функціональної стійкості розподіленої програмної системи для систем електронної комерції.

2 МОДЕЛІ ЗАБЕЗПЕЧЕННЯ ФУНКЦІОНАЛЬНОЇ СТІЙКОСТІ РОЗПОДІЛЕНИХ ПРОГРАМНИХ СИСТЕМ

2.1 Загальний підхід до забезпечення функціональної стійкості

Ми пропонуємо структуру для сприяння розвитку програмних систем забезпечення функціональної стійкості на основі середовища імітаційного моделювання. Розділ складається з загальної моделі забезпечення функціональної стійкості, механізму складання моделей для інтеграції базових моделей (тих, що задають базову функціональність системи) та моделей забезпечення функціональної стійкості, а також платформи генерації коду, яка підтримує автоматизовану генерацію програмних систем з підтримкою забезпечення функціональної стійкості. Метою цього середовища є інтеграція різних існуючих методів забезпечення функціональної стійкості (таких, як зазначені у розділі 2), з аналогічними моделями архітектури (MDA)[9], з метою створення єдиної платформи для забезпечення програмного забезпечення забезпечення функціональної стійкості систем, що розробляються та впроваджуються з наявних технічних специфікацій.

2.2 Модель забезпечення функціональної стійкості

На рисунку 2.1 показаний фрагмент моделі забезпечення функціональної стійкості компонентів та сервісів. Для різних типів програмного забезпечення, таких як компонент програмного забезпечення або веб-служба, можуть бути різні типи несправностей. Збій програмного забезпечення може бути виявлений за допомогою одного або декількох детекторів та усунено за допомогою одного або декількох механізмів розподілу програмних компонент за обчислювальними ресурсами.

Вирішення несправності моделюється набором дій, які перетворюють систему зі свого несправного стану у визначений робочий стан. Так само можна вжити профілактичні дії, коли прогнозується виникнення несправностей.

Модель забезпечення функціональної стійкості визначає категорії несправностей, які можуть виникнути для компонентів та служб: синтаксичні помилки, семантичні помилки, сервісні помилки, помилки комунікації / взаємодії та винятки. Кожен тип несправностей використовує свої специфічні детектори/предитектори та механізми усунення несправностей. Кожен тип несправностей також може бути розбитий на більш конкретні несправності залежно від характеру компонента та області застосування.

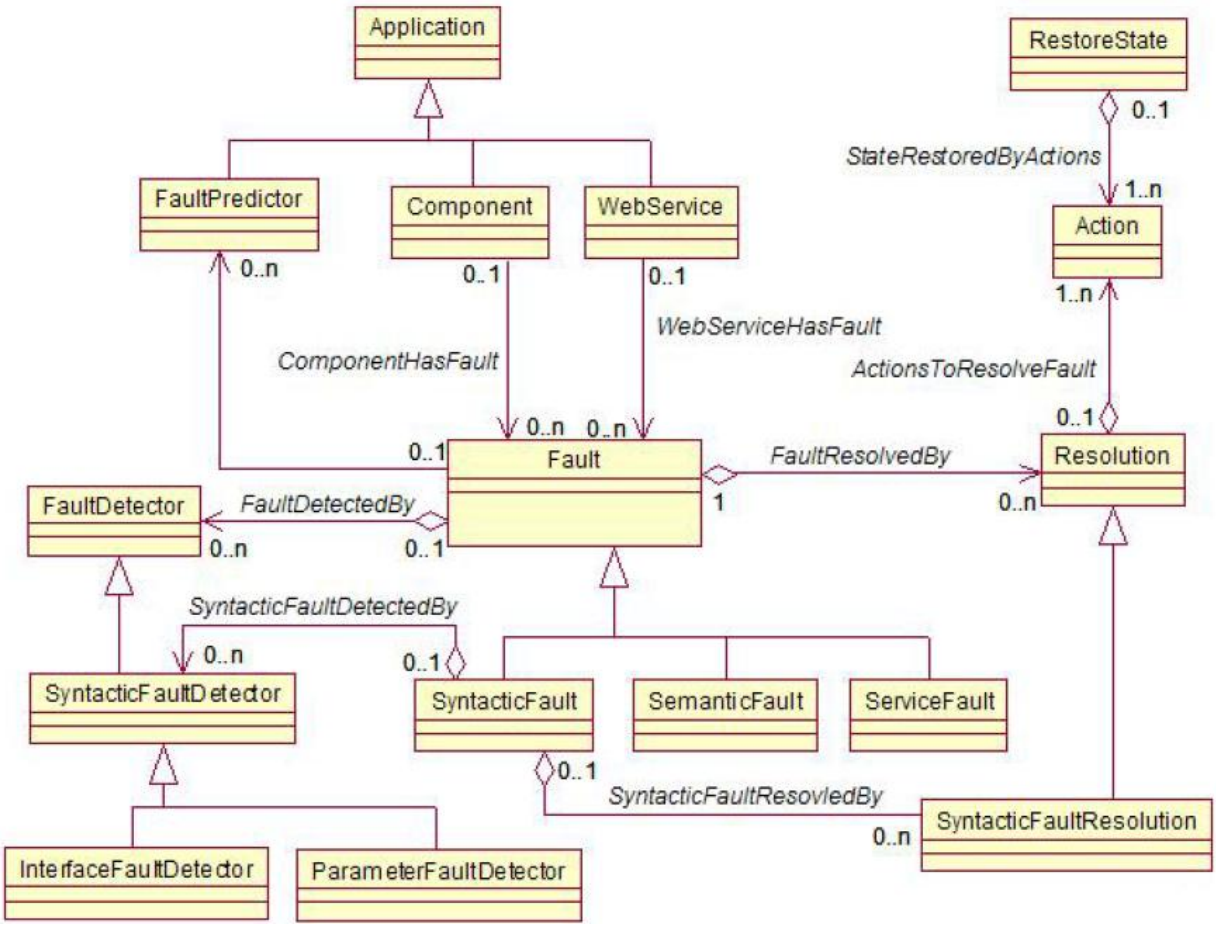

Рисунок 2.1 – Фрагмент моделі забезпечення функціональної стійкості

2.2 Структурна модель забезпечення функціональної стійкості

Модель забезпечення функціональної стійкості описується як сукупність абстрактних конструкцій, які інтенсифікуються, коли вона асоціюється з базовими моделями системи. Таким чином, функція забезпечення функціональної стійкості зберігається окремо від базової функціональності програмної системи. Розкладання базової функціональності та особливості забезпечення функціональної стійкості покращує модульність та повторну використання модулів забезпечення функціональної стійкості. Основна перевага цього підходу полягає в тому, що він дозволяє розробляти та підтримувати моделі забезпечення функціональної стійкості незалежно від базової моделі. Крім того, моделі забезпечення функціональної стійкості можуть застосовуватися на всіх рівнях в ієрархії програмних систем. Рисунок 2.2 ілюструє сценарій, в якому різні моделі забезпечення функціональної стійкості (позначені пунктирними прямокутниками) можуть складатися з різних програмних модулів, від рівня класу, рівня компонентів до системного рівня.

Встановлення моделі забезпечення функціональної стійкості в базову модель програмної системи застосування здійснюється методами композиційного моделювання. Зазвичай, композиція моделі передбачає об’єднання двох або більше різних моделей для отримання єдиної інтегрованої моделі.

Аспектно-орієнтоване моделювання (АОМ)[10] є перспективною дослідницькою областю, яка підтримує склад моделі. Моделі самолікування можуть бути визначені як аспекти перехресної дії, які вбудовані в базові моделі. Детектори несправностей призначені для фіксації аномального статусу програмного забезпечення (що може піти не так, а також коли і де воно пішло не так). Ці зразки можна позначати за допомогою дескрипторів точкового вказування.

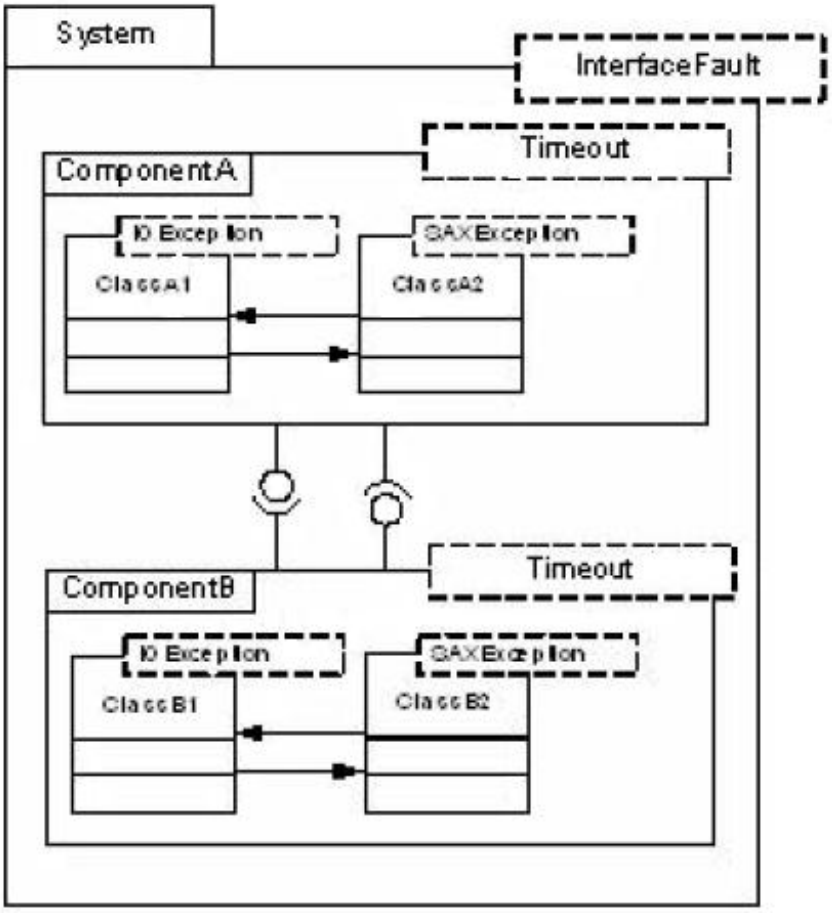

Рисунок 2.2 – Склад базових моделей та моделей забезпечення функціональної стійкості

Постанови - це дії, які будуть вжиті, коли станеться певний тип відмови. Ці дії можуть бути капсульовані в рамках розробленої аспектної моделі. Перетин аспектів створює моделі аспектів і прив’язує їх до базових моделей. Модельне зборки [11] - це ще один підхід до складання моделі, де відносини між моделями фіксуються під час зборки програмної системи. Модель зборки - це модель, яка визначає різні види відображень між елементами моделі. Застосовуючи цей підхід, створюється окрема модель зборки, яка чітко визначає зв’язки між базовими моделями та моделями моделей забезпечення функціональної стійкості. Усі три види моделей (тобто базових, аспектно-орієнтованих і моделей зборки) передаються в середовище виконання та складатимуться в єдину інтегровану модель.

3 МЕТОД ЗАБЕЗПЕЧЕННЯ ФУНКЦІОНАЛЬНОЇ СТІЙКОСТІ ПРОГРАМНИХ СИСТЕМ

3.1. Основні етапи методу забезпечення функціональної стійкості розподілених програмних систем

Моделі забезпечення функціональної стійкості конкретизуються незалежно від платформи. Для підтримки повної програми забезпечення функціональної стійкості необхідно використовувати сімейство генераторів кодів [12] для перетворення незалежних від платформи моделей у специфічний для платформи код, кожен з яких описує, як функції забезпечення функціональної стійкості реалізуються на іншій платформі.

Як показано на рисунку 3.1, основні та моделі забезпечення функціональної стійкості переводяться відповідно до платформних реалізацій власними генераторами коду відповідно. Зокрема, представлення несправностей програмного забезпечення та їх рішень, що реагують на них, перетворюються на код реалізації, який виявляє виникнення несправностей та вживає відповідних дій для відновлення системи до заданого робочого стану. Потім базовий код доповнюється інструментом самолікування, використовуючи методи складання програм [13]. Як результат, побудована повна програмна система, що підтримує самолікування, і безпосередньо відображається на представлення складеної моделі програмного забезпечення з можливістю забезпечення функціональної стійкості, на основі якої може бути проведений аналіз на з використанням верифікації та перевірки системи.

Розглянемо основні етапи запропонованого методу:

- композиція базової моделі та моделі забезпечення функціональної стійкості системи;

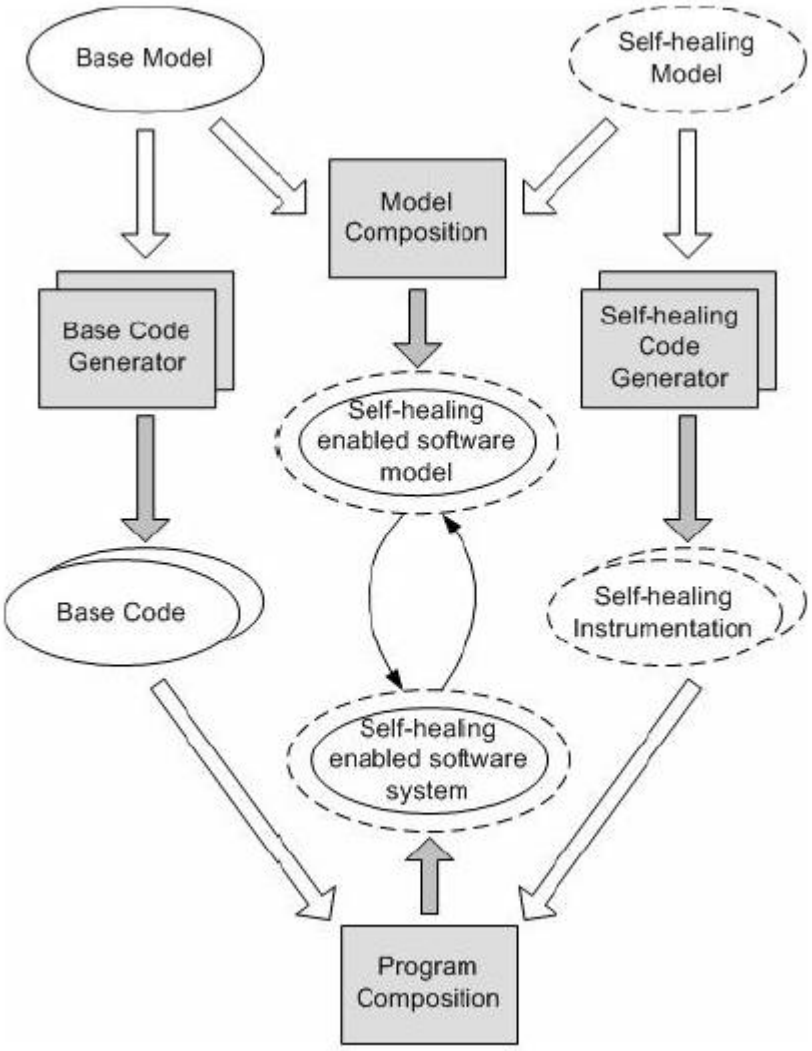

Рисунок 3.1 – Послідовність генерації програмних компонент для забезпечення функціональної стійкості

- паралельна генерація коду базової моделі та моделі забезпечення функціональної стійкості;

- запуск базової програмної системи та програмної моделі забезпечення функціональної стійкості;

- забезпечення відстеження та порівняння станів програмної системи та її моделі;

- при виявленні розбіжностей в функціонуванні програмної системи та її моделі активувати модель забезпечення функціональної стійкості;

- використати інструменти забезпечення функціональної стійкості згідно з результатами моделювання на етапі 5;

- якщо програмна система продовжує виконання, перейти до етапу 4., якщо ні – завершити роботу усього програмного забезпечення.

3.3. Місце розробленого методу у дослідженнях сучасних програмних систем

Стандартна класифікація IEEE для аномалій програмного забезпечення [3] пропонує повний перелік класифікації програмних аномалій та пов’язані з ними елементи даних, які корисні для виявлення та відстеження аномалій. Методологія цього стандарту базується на методі (послідовності етапів), який здійснює логічний прогрес від початкового розпізнавання аномалії до їх остаточного усунення. У [6] запропонована систематика несправностей компонентів програмного забезпечення для полегшення ідентифікації та класифікації поширених несправностей у компонентному програмному забезпеченні. Автори [5] описують систематику відмов, помилок та несправностей для надійних та безпечних обчислень. Засоби досягнення надійності та безпеки класифікуються як запобігання несправності, толерантність, усунення несправностей та прогнозування несправностей. В [14], Для класифікації програмних відмов на основі профілів виконання запропоновані методи на основі дерев. Профіль UML для моделювання QoS та відмовостійкості [15] визначає набір розширень UML для представлення концепцій QoS та відмов на основі реплікацій об’єкта.

Проекти в програмі DARPA DASADA [16] [17] описують архітектуру, яка використовує зонди та датчики для контролю за виконанням програм, генерування подій, що містять вимірювання, та вжиття дій на основі інтерпретації подій. Ефектори взаємодіють із системою, щоб переконатися, що поведінка під час виконання системи вписується в рамки прийнятної поведінки. Автори в [18] пропонують генерацію проксі-серверів і обгорток, щоб додати автономних функціональних можливостей об’єктно-орієнтованим додаткам, щоб впоратися з відмовами без адаптації вихідного

коду. У [19] автори описують використання перетворення коду для інструменту байт-коду Java, додаючи аналіз несправностей та дії з лікування. Автори [20] представляють механізм самолікування програмних компонентів на основі з’єднувачів. Компонент в системі самолікування розроблений у вигляді шаруватої архітектури, структурованої з цілющим шаром та службовим шаром. Роль з’єднувачів між завданнями в компоненті розширюється для підтримки механізму самолікування у виявленні, перенастроюванні та ремонті аномальних об’єктів у компонентах.

Новизна нашого підходу до програмних систем забезпечення функціональної стійкості лежить у моделюючій системі інтеграції функціональних та самолікувальних специфікацій, а також у застосуванні трансформації моделі для реалізації можливостей забезпечення функціональної стійкості. Наші рамки моделювання мають на меті бути більш загальними, ніж попередні підходи, оскільки ми синтезуємо різні методи самолікування та застосовуємо усі моделі та механізми трансформації для підтримки повноцінного програмного забезпечення для забезпечення функціональної стійкості від абстрактних специфікацій моделей аж до впровадження системи.

4 ВПРОВАДЖЕННЯ ЗАПРОПОНОВАНОГО МЕТОДУ ЗАБЕЗПЕЧЕННЯ ФУНКЦІОНАЛЬНОЇ СТІЙКОСТІ В ЕЛЕКТРОННІЙ КОМЕРЦІЇ

4.1 Особливості забезпечення функціональної стійкості систем електронної комерції

Зважаючи на високу швидкість банківської транзакції, це важливо для концепції електронного банкінгу, що програмне забезпечення та апаратної практичної пропускної здатності повинно бути достатньо. Таким чином, система може бути досягнута до максимальної доступності, і тому клієнти ніколи не втрачають довіри до системи. Оскільки для них важливіше звертатись до кожної банківської послуги, вона працює правильно.

Використовуючи цикл забезпечення функціональної стійкості в системі, SOA (Service Oriented Architecture) має найкращі відгуки серед сучасних архітектур. Оскільки кожна частина системи розглядається як послуга, то керування та виявлення несправностей буде простішим за інші архітектури, такі як CBA (Component Based Architecture) або AOA (Aspect Oriented Architecture). Але модель електронного банкінгу більше половини банків базується на централізованому банківському бізнесі, елементи системи якого керуються централізованим рівнем важливості безпеки. Краще оновити ці моделі до SOA. Рисунок 4.1показує наступну схему банку.

Отже, відповідно до верхньої схеми, ми можемо класифікувати відмови (рисунок 4.2). Зважаючи на цю класифікацію, збої, які трапилися в банківській службі, розділені на дві категорії:

Провал бізнесу: збої, які трапляються для помилок користувачів або бізнес обмежує час трансакції.

Поломка системи: деякі збої, що випадають з мережевої інфраструктури, комутаторів, підключення до централізованої системи або

підключення до іншого каналу, наприклад SHETAB7 та інтерфейсних комутаторів.

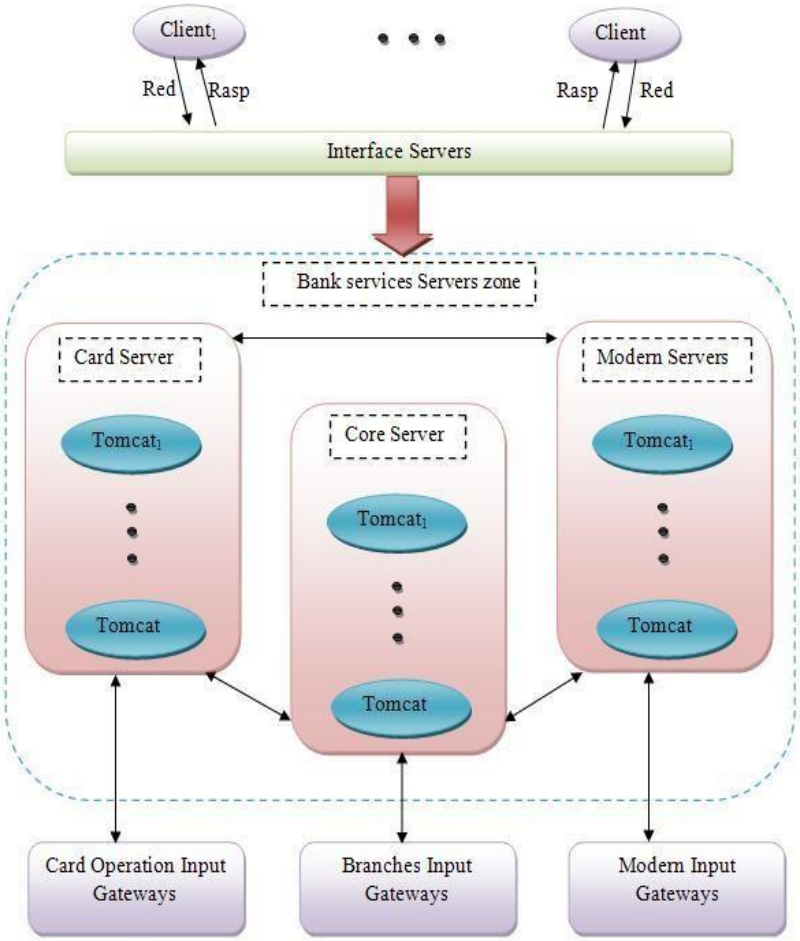

Рисунок 4.1 – Схема електронного банкінгу

Зауважимо, що знання клієнтів щодо електронного банкінгу в розвинених країнах більше, ніж у країнах, що розвиваються. Отже збої в бізнесі частіше, ніж збої в роботі системи. У таблиці 4.1 показано приклад станів транзакцій з помилками.

Таким чином, доступність є важливою умовою перевірки відмов. Наприклад, міжбанківські комутатори не могли зрозуміти, що вони не працюють. Тож якщо доступність комутаторів більша, то представлена статистика з точки зору вимикачів мають більшу надійність. Обчислює доступність за трьома індексами: внутрішніми, емітентними, отримують, що

результат їх показаний у таблиці II. Примітно, що наша мета полягає не лише в тому, щоб зменшити недоліки, які призводять до банківських послуг, недоступними.

Відповідно до статистичних даних, метою даної роботи стане підвищення доступності шляхом мінімізації системних збоїв. Ці несправності виявлені експертними особами, і вони усунуть ці виявлені несправності. Після цього, коли вони випробували загоєні компоненти, відмовившись від роботи та переконавшись у своїй практиці здоров’я, повертають їх до системного циклу, який показує результат моніторингу транзакцій за згаданий період, який включає кількість та відсотки транзакцій, що призводять до помилок.

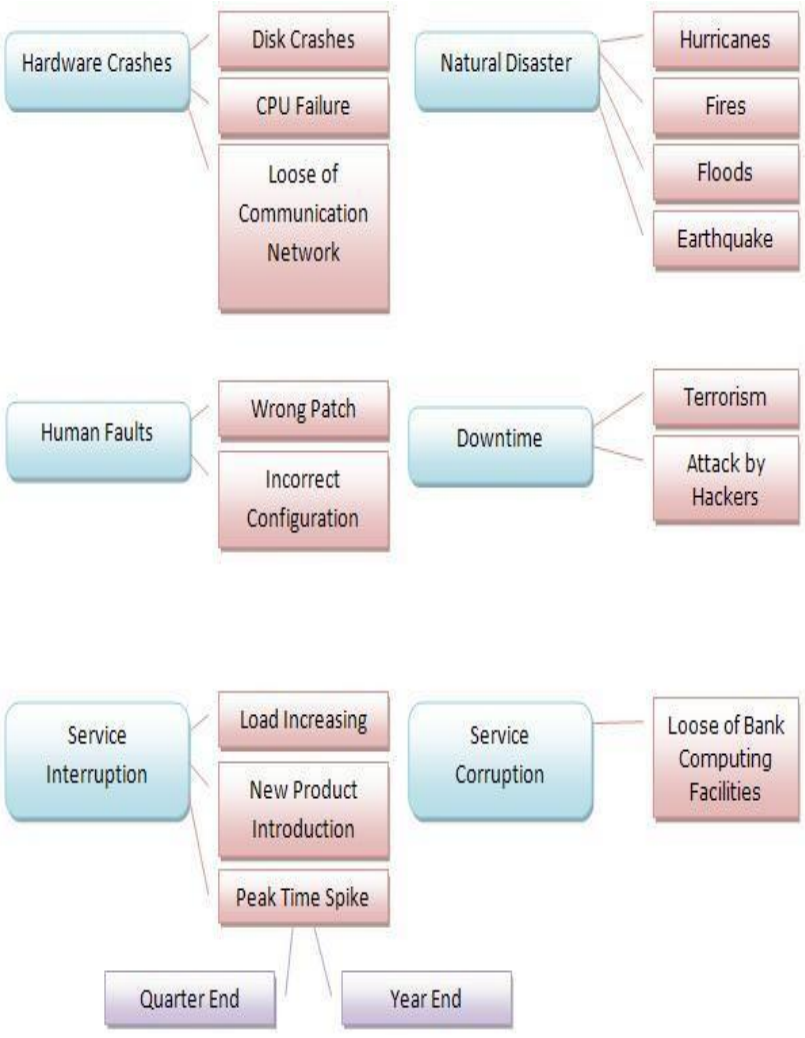

Рисунок 4.2 – Класифікація відмов

Таблиця 4.1 – Статус трансакцій з помилками

Відповідь Код	Текст відповіді	Кількість транзакцій
6	Помилка	27
12	Недійсна транзакція	61977
25	Неможливо знайти запис у файлі	13756
30	Помилка форматування	985
31	Банк не підтримується комутатором	36
68	Відповідь надійшла надто пізно	57832
77	Slm - недійсна дата бізнесу	9881
80	Слім-транзакцію відхилено	14774
84	Slm-емітент вниз	284640
87	Помилка синхронізації ключа	2138 рік
90	Відсікання триває	35799
91	Емітент або перемикач не працює або знижується	301366
93	Транзакція не може бути завершена	41220
94	Дублікат передачі	903
96	Несправність системи	49493

Лише 3% транзакцій призводять до помилок системи, що деякі з них можуть бути збоєм. Близько 0,02 цих транзакцій перебувають у режимі емітента, що нам ніколи не потрібно знати, що це; тому що ці помилки повинні враховувати банки-емітенти, і ми збираємося усунути несправності, які можуть призвести до збоїв у придбанні та внутрішньому режимі. Таким чином, використовуючи знання експертних осіб, ми створили невелику таблицю відмов.

4.2 Реалізація етапів методу забезпечення функціональної стійкості розподілених програмних систем підтримки електронної комерції

Далі ми описуємо етапи самолікування в нашому прикладі з наступних розділів.

4.2.1. Моніторінг

Задля реалізації моніторінгу, зауважимо, що різні архітектури реалізують різні методи збору необхідних даних для фази виявлення збоїв. Зважаючи на проблемну модель, ми використовували метод збору даних із системи за допомогою визначення деяких показників; спочатку ми визначаємо системні атрибути та зберігаємо їх у базі даних, яка називається база даних атрибутів. Потім ми визначаємо деякі показники відповідно до системних вимог та знань експертних осіб. Ми називаємо їх "системні метрики" і використовуємо атрибути з бази даних атрибутів для визначення метрики. Для використання цих показників у частинах системи потрібно визначити деякі метричні протоколи, наприклад:

$$\{M \mid M \in \text{Metrics AND} \exists \text{ Attributes AND} \exists C \in \text{Conditions} : (A : C)\}$$
$$\{p \mid p \in \text{Protocols AND} \exists M \in \text{Metrics AND} \exists \ sc \in sys_{spnt} : (m, \{sc\})\}$$

Для цих значень, наприклад, ми маємо: атрибути: $\{a_1, a_2, \ldots a_n\}$; умови: $\{c_1, c_2, \ldots, c_m\}$; метрики: $\{M_1: \{a_1: c_1; a_2: {}_{c2}; a_3: c_3\}, \ldots, M_l: \{a_2: c_6; a_{n-4}: c_4; a_n: c_9\}\}$.

$$Sys_{cpnt}: \{sc_1, sc_2, \ldots, sc_k\},$$

де $c_1, c_2, \ldots, c_m$ є умовами, які присвоюють деяким атрибутам, а sys_{cpnt} - це список системних компонентів, які мають застосовувати до їхньої метрики. Потім протоколи були визначені нижче. Протоколи: p_1 (m_1, $\{sc_3$,

sc_{11}, sc_{k-1}}), p2 (m_4, {sc_1, sc_8, sc_k},…, p_2 (m_{l-1}, {}).

Якщо для кожного протоколу системні компоненти встановлені нульовими, то це означає, що для всіх компонентів системи слід застосовувати стандартну метрику. Зважаючи на специфіку комерційних систем, ми можемо зіставити згадані елементи наступним чином:

Атрибути:

- час відгуку сервера для пакетів з піском;
- час відповіді на базу даних;
- точність обчислювальної ефективності банківських послуг;
- відправте клієнтам повідомлення про відсотковий депозит.

Показники:

- час відгуку сервера для пакетів з піском не повинно перевищувати 2 секунди;
- час відповіді сервера бази даних не повинно перевищувати 1 секунди;
- послуга tell-bank не повинна бути призначена законному клієнту;
- кожен номер мобільного телефону повинен присвоювати лише одному клієнту.

Метричні протоколи:

- кожен номер мобільного телефону повинен присвоювати лише одному клієнту: {Мобільний банк на основі USSD};
- час відповіді сервера для пакетів з піском не повинно перевищувати 2 секунд: {};
- надішліть повідомлення про відсотковий депозит на суму менше 1 000 000 R: {CHAPAR 8, служба моніторингу творців повідомлень}.

Коли наші визначення будуть завершені, ми встановлюємо деякі компоненти, які включали кілька методів використання цих протоколів. Ми називаємо їх екологічними компонентами, які зобов’язані збирати дані з системи, а рішення про те, які з цих даних можуть бути корисними для захищених показників, які використовуються в протоколах, буде зроблено на

наступному етапі. Наявні компоненти відповідають визначеним періодам планів та політик організації. Наприклад, ми можемо визначити наступні компоненти (таблиця 4.2).

Отже, поки збирання даних завершено, журнали даних переносяться на фазу виявлення.

Таблиця 4.2 – Приклад компонентів системи

Назва компоне-нта	Метод	Параметр	Опис
Замовник	Customer_Info ()	CID	Отримайте тестову інформацію про клієнта за його / її ідентифікатором
Замовник	Customer_Info ()	ІДНО	Отримайте тестову інформацію про клієнта від його / її ІДНО
Замовник	Customer_Deposit_ Info ()	CID	Отримайте інформацію про депозит для тестового клієнта за його особою
Замовник	Customer_Deposit_ Info ()	DNO	Отримайте інформацію про депозит для клієнта-тестувальника через його депозит число
Повідом-лення	Send_Deposit()	CID	Надішліть повідомлення з меншою сумою ніж 1 000 000R для замовника тесту
Мережа	Telnet ()	Node_IP	Трансляція пакетів telnet для вузлів та отримання їх відповідь

4.2.2. Виявлення збоїв та відмов

Дані, зібрані з попереднього етапу, інтерпретуються різними методами та визначають критичні випадки. V-Cure застосовує базу даних, що складається з нормального стану системи для кожного компонента і порівнює результати моніторингу з цим статусом та визначає підозрілі випадки, які можуть призвести до збою. Але стосовно масштабу комерційної системи ми застосовуємо метричні протоколи. Тож після виявлення частин системи, які мають такі умови, як стан відмови, ми повинні застосувати

методи їх лікування. Але, враховуючи кількість виявлених випадків і малу важливість більшості з них, необхідно кластеризувати ці випадки, а потім кластери з високим ступенем важливості вибираються для застосування дій відновлення.

$$\{C \mid C \in \text{Clasters AND} \exists A, B \in \text{Attributes} : (C : A, B)\}$$

Слід зазначити, що коефіцієнт ступеню кластеризації визначається експертними особами. Для нашого конкретного випадку ми можемо об’єднати моніторинг доказів наступним чином:

- C1: розрахунок відсотків за довгостроковими депозитами.
- C2: затримка часу на надсилання та отримання повідомлень, затримка в часі підключення до бази даних.
- C3: надіслати повідомлення про відсотковий депозит менше 1000000 клієнтам, які є одним із членів групи з обмеженими умовами.

Після відображення даних моніторингу у відповідний кластер, кластери з більш високим ступенем значення переходять у фазу авідновлення функціональної стійкості, щоб зробити відновлення на них.

4.2.3. Відновлення функціональності

Після виявлення відмови на фазі виявлення настає черга загоєння цих невдач діагнозу. Зважаючи на попередні розділи, ми можемо знайти можливе рішення, використовуючи таблицю відмов, і компонент можна оздоровити. Отже, існують різні підходи до досягнення цієї мети, деякі з яких перелічені нижче. Більшість уже існуючих розчинів для самолікування відрізняються міжзаступними механізмами, які вони використовують. Отже, є деякі методи відновлення несправностей, перелічені нижче.

Методи самолікування та відновлення. Деякий механізм самозбірки, заснований на об’єкті агента, який реплікує компоненти для заміни

відмививших сусідів та дозволяє перекомпонувати цілі структури. Інша стратегія, натхненна біологією, полягає у наданні системі можливості реплікації клітин надлишком для боротьби із зовнішніми вторгненнями [8]. Один із методів, що застосовуються в обчислювальних процесах, орієнтованих на відновлення (ROC), полягає у виділенні несправних компонентів та заміні їх зайвими [6].

Моделі та політики архітектури.

Деякі середовища, що базуються на компонентах, підтримують взаємозамінні архітектурні стилі відповідно до відхилень продуктивності. Два приклади - [7] та [8]. Перший статично асоціює набір правил дії для кожної з попередньо визначених причин відмови. Другий використовує деякі функції утиліти, щоб вибрати найбільш підходящий архітектурний варіант для усунення несправності. Ви можете побачити інший тип архітектурних моделей, який представили дослідники у [9-12].

Мікроперезавантаження компонента.

Несправні модулі мікрозавантажуються незалежно та автоматично, щоб уникнути поширення несправностей, коли їх підозрюють, що вони не працюють належним чином. Ефективність цієї методики полягає в тому, що перезапуск окремих компонентів займає менше часу, ніж перезавантаження всієї системи [13, 14].

Існують і інші дії з відновлення, такі як реорганізація процесів на основі SOA [15], методи голосування. Поважаючи нашу модель, ми використовуємо першу модель для відновлення несправностей. Перед будь-якою дією ми повинні створити таблицю несправностей. Якщо не існує жодного рішення щодо несправності, система може повідомити про це експерту. Тоді він може додати його до таблиці несправностей, щоб система могла впоратися з цим збоєм, якщо він стався пізніше.

4.2.4. Тестування після відновлення

Після відновлення дії компонента, який призводив до відмови, ми використовуємо певний тестовий сценарій, щоб переконатися у відсутності негативного впливу на інші компоненти. Якщо результат тестування нормальний, відновлений компонент буде повернутий у систему; в іншому випадку система повинна повідомити систему управління.

Зважаючи на цю платформу, система може запустити відновлення щодо кожної несправності, що призводить до відмови, і цим рішенням; немає необхідності в експерті для деяких збоїв, які можуть відновити невдачу за допомогою циклу забезпечення функціональної стійкості.

4.3. Експерементальні дані використання системи забезпечення функціональної стійкості

У цьому підрозділі ми порівнюємо наш метод з тим, що використовувався в системах електронної комерції раніше. Тепер припустимо усунення дефектів, які використовуються системами придбання та внутрішніми системами, які також використовуються системами емітентів. Тож збої зменшаться з обох сторін, і ми можемо отримати систему з високою доступністю.

Якщо ми досліджуємо систему за таблицею несправностей, яка наведена вишче, в архітектурі забезпечення функціональної стійкості, то, порівняння кількісті збоїв, показано на рисунку 4.3.

На рисунку 4.4 показаний результат при оптимізованому стані. Якщо врахувати якийсь інший фактор, то кількість виявлених збоїв, які були вилікувані, зміниться.

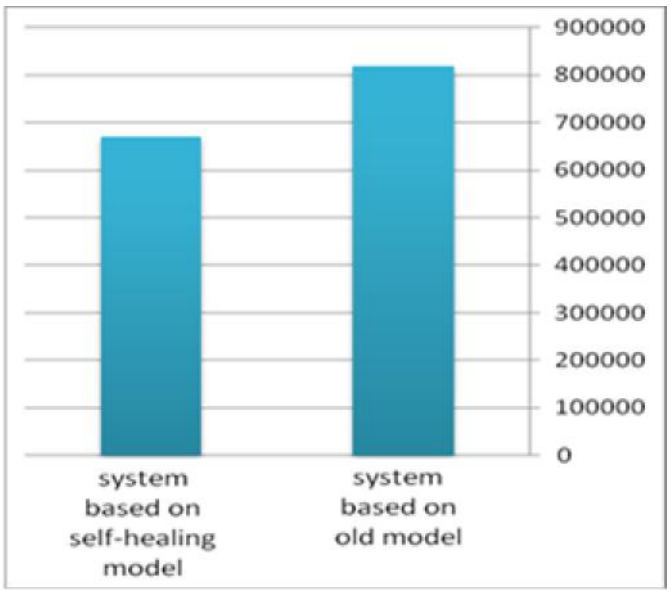

Рисунок 4.3 – Порівняння кількісті невдач до та після використання методу забезпечення функціональної стійкості

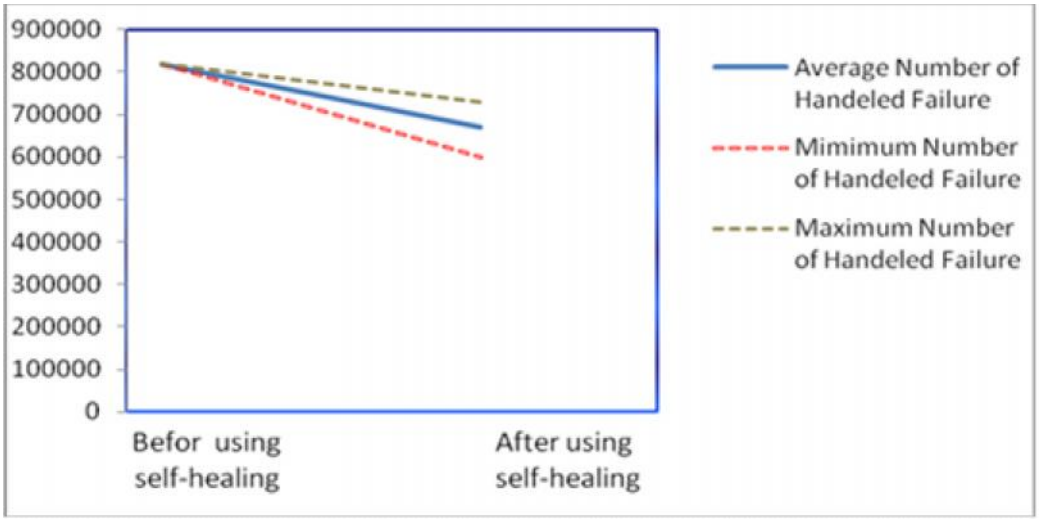

Рисунок 4.4 – Зміна коефіцієнту кількості відмов системи

Наприклад, різні послідовності несправностей призводять до відмови, і лише деякі з них були ідентифіковані та використані для показників на фазі виявлення. Тому невдача повинна бути визначена експертними особами та методами розпізнавання шаблонів та додана до показників.

ВИСНОВКИ

У магістерській атестаційній роботі представлено загальну основу для моделювання програмного забезпечення самолікування. Конструкти моделювання використовуються для збору несправностей програмного забезпечення та їх виявлення та вирішення для перетворення системи з несправних станів в робочі стани, зазначені в моделі. Класифікація програмних несправностей та моделювання виявлення та усунення несправностей сприяють побудові програмних систем забезпечення функціональної стійкості. Забезпечення функціональної стійкості досягається шляхом перетворення моделей в конкретні платформи, які потім складаються з базового модуля для формування інтегрованої програмної системи, яка надає рішення про відмову, щоб зменшити вплив програмних несправностей.

Для досягнення загальносистемної оптимізації, окрім забезпечення функціональної стійкості, необхідна політика для того, щоб організувати поведінку системи та налагодити її діяльність відповідно до бізнес-цілей. Для досягнення автономних обчислень необхідно інтегрувати різні аспекти забезпечення функціональної стійкості. Це буде основою майбутніх досліджень, які також досліджуватимуть інтеграцію онтологічних та виробничих правил для покращення моделювання несправностей з можливістю штучного інтелекту для поліпшення аналізу та класифікації несправностей.

У своїй роботі також ми аналізуємо комерційні середовища електронної комерції та класифікуємо помилкову поведінку цих систем. Тож ми представили новий підхід, заснований на забезпечення функціональної стійкості. Додавши цикл відновлення, ми використали певну дію відновлення невдач, і результат показав, що ми можемо скоротити час повернення системи до безпечного режиму. Таким чином, ми могли б мати більш надійні системи, ніж системи, що не використовують засоби забезпечення

функціональної стійкості. Ми використали кілька метрик та метричних протоколів для ідентифікованих системних частин, які мають проблеми, а потім кластеризували їх. Таким чином, наша система застосовує відновлення для виявленого збою. Але, незважаючи на все це, ми не змогли позбутися експертів в системі. Майбутня робота буде включати інтеграцію прогнозування відмов з використанням методології навчання та дій для відновлення невдач. Також ми можемо збільшити оптимізацію прогнозування шляхом виявлення знань про збір несправностей.

Таким чином, мета роботи, а саме збільшення надійності та відмовостійкості розподілених програмних систем шляхом розробки моделі та методу забезпечення функціональної стійкості була досягнута.

Вирішили наступні задачі:

- досліджено існуючі засоби забезпечення надійності та відмово стійкості розподілених програмних систем;
- розроблено модель забезпечення функціональної стійкості розподіленої програмної системи;
- розроблено метод забезпечення функціональної стійкості розподіленої програмної системи;
- розроблено макет програмних засобів підтримки функціональної стійкості розподіленої програмної системи для систем електронної комерції.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. John Strassner and Jerey O. Kephart, \Autonomic Systems and Networks: Theory and Practice, Network Operations and Management Symposium (NOMS), 2016.

2. Jerey O. Kephart and David M. Chess, The Vision of Autonomic Computing, IEEE Computer, Vol. 36(1), January 2013.

3. IEEE Standard Classication for Software Anomalies, IEEE Std 1044-1993, 1993. A Modeling Framework for Self-Healing Software Systems 9.

4. John Musa, Software Reliability Engineering, McGraw-Hill, 1999.

5. Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, Carl Landwehr, \Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing, Volume 1, Issue 1, 2014, pp. 11-33.

6. Leonardo Mariani, A fault taxonomy for component-based software, Proceedings of International Workshop on Test and Analysis of Component-Based Systems (TACoS), Electronic Notes in Theoretical Computer Science, Vol. 82, Elsevier Scence, 2013.

7. Gregor Kiczales, et al, Aspect-Oriented Programming, European Conference on Object-Oriented Programming, Finland, June 2011.

8. Salim Hariri, Alok Choudhary, and Behcet Sarikaya, Architectural Support for Designing Fault-Tolerant Open Distributed Systems. IEEE Computer, 1992.

9. Richard Soley and the OMG Sta Strategy Group, Model-Driven Architecture, [Електронний ресурс] ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf.

10. Didonet Del Fabro Marcos, Bzivin Jean, Jouault Frdric, Breton Erwan, Gueltas Guillaume, AMW: A Generic Model Weaver, In Proceedings of the 1re Journesur l'Ingnierie Dirige par les Modles (IDM05), Paris, France, June-July

2015.

11. Markus Voelter, A Collection of Patterns for Program Generation, Eighth European Conference on Pattern Languages of Programs, Irsee, Germany, June 2013.

12. Uwe Amann, Invasive Software Composition, Springer-Verlag, 2003.

13. Patrick Francis, David Leon, Melinda Minch, Andy Podgurski, Tree-Based Methods for Classifying Software Failures, Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04), Saint-Malo, France, November 2014.

14. Object Management Group, UML Pro le for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, [Електронний ресурс] http://www.omg.org/cgibin/doc?formal/06-05-02, 2016.

15. David Garlan, Bradley Schmerl and Jichuan Chang, Using Gauges for Architecture-based Monitoring and Adaptation, Working Conference on Complex and Dynamic Systems Architecture, Australia, 2011.

16. Janak Parekh, et al, Retrotting autonomic capabilities onto legacy systems, Journal of Cluster Computing, 2005, pp. 141-159.

17. A. Reza Haydarlou, et al, A Self-healing Approach for Object-Oriented Applications, 3rd International Workshop on Self-Adaptive and Autonomic Computing Systems, 2005.

18. M. Muztaba Fuad and Michael J. Oudshoorn, Transformation of Existing Programs into Autonomic and Self-healing Entities, 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), 2007.

19. Michael E. Shin and Daniel Cooke, Connector-Based Self-Healing Mechanism for Components of a Reliable System, Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005), St. Louis, Missouri, May 21, 2005.

20. F. Calisir, C. A. Gumussory, Internet Banking versus Other Banking Channels: Young Consumers' View, International Journal of Information

Management, pp 215-221, 2018.

21. H. A. Al-Zu'bi, A. M. Ahmad, E-banking Functionality and Outcomes of Customer Satisfaction: An Empirical Investigation, International Journal of Marketing Studies Vol. 3, pp. 50-65 February 2011.

22. M. Xue, L. M. Hitt, P. Y. Chen, The Determinants and Outcomes of Internet Banking Adoption, Management Science (Forthcoming), 2011.

23. B. Goodwin-Jones, Emerging technologies – accessibility and web design why does it matter? Language Learning and Technology, pp. 11-19, 2001.

24. M. O. Hilari, Quality of Service (QoS) in SOA Systems: A Systematic Review, 2009.

25. D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl, P. Steenkiste, Rainbow: Architecture-based self adaptation with reusable infrastructure, IEEE Computer, vol. 37, pp. 46-54, October 2014.

26. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, Using architecture models for runtime adaptability, IEEE Software, vol. 23, pp. 62-70, 2006.

27. E. M. Dashofy, A. v. Hoek, R. N. Taylor, Towards architecture-based self-healing systems, in WOSS'02: Proceedings of the first workshop on Self-healing systems, New York, NY, USA, pp. 21-26, 2012.

28. M. E. Shin, J. H. An, Self-reconfiguration in self-healing systems, in EASE'06: Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems, Washington, DC, USA, pp. 89-98, 2006.

29. S. Montani, C. Anglano, Achieving self-healing in service delivery software systems by means of case-based reasoning, Applied Intelligence, vol. 28, pp. 139-152, 2008.

30. M. M. Fuad, M. J. Oudshoorn, Transformation of existing programs into autonomic and self-healing entities, in ECBS'07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, Washington, DC, USA, pp. 133-144, 2007.