МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра комп'ютерної інженерії

СКЛАДАНЮК Владислав Миколайович

Програмний засіб синтезу моделей акторів на основі предметно-орієнтованого підходу / Software for Synthesis of Actor Models Based on Subject-Oriented Approach

спеціальність: 123 – Комп'ютерна інженерія
освітньо-професійна програма – Комп'ютерна інженерія

Кваліфікаційна робота

Виконав: студент групи КІ-41
Складанюк Владислав Миколайович
_____

Науковий керівник
К.т.н., старший викладач Савка Н.Я.

Кваліфікаційну роботу
Допущено до захисту
«___» _____ 20 ___ р.

Завідувач кафедри
_____ О.М. Березький

ТЕРНОПІЛЬ - 2021

РЕЗЮМЕ

Кваліфікаційна робота на тему «Програмний засіб синтезу моделей акторів на основі предметно-орієнтованого підходу» зі спеціальності 123 «Комп'ютерна інженерія» освітнього ступеня «бакалавр» містить 85 сторінок пояснюючої записки, 13 рисунків, 8 таблиць, 2 додатки. Обсяг графічного матеріалу 2 аркуші формату А3.

Метою кваліфікаційної роботи є розробка програмного засобу синтезу моделей акторів в межах предметно-орієнтованого підходу.

Розглянуто задачу організації даних у розподіленому середовищі. Для розв'язування таких задач існують методи, алгоритм та програмні засоби, в основі яких паралельні обчислення. Такі засоби вимагають складних обчислювальних процедур й при збільшенні даних програть у продуктивності.

Досліджено моделі реалізації розподілених систем, які гуртуються на окремих об'єктах – акторах, що прості у реалізації, а їх функціонування базується на надсиланні повідомлень між акторами. Розглянуто програмні засоби реалізації моделей акторів й відзначено фреймворк Akka

Розроблено алгоритм синтезу моделей акторів на основі фреймворку Akka та візуального підходу, що враховує усі властивості моделей, відзначається швидкодією та постотою.

Побудовано розподілену систему на основі акторної моделі для організації даних із спільною пам'яттю.. Проведено її програмну реалізацію та результатами експериментів підтверджено її ефектиність.

Програмна система синтезу моделей акторів на основі візуального підходу містить додатковий рівень абстракцій та досить ефективна при реалізації паралельних та розподілених обчислень.

Ключові слова: МОДЕЛЬ АКТОРІВ, ПРЕДМЕТНО-ОРІЄНТОВАНИЙ ПІДХІД, РОЗПОДІЛЕНА СИСТЕМА, ФРЕЙМВОРК AKKA.

RESUME

Qualification thesis “Software of Synthesis of Actor Models Based on Subject-Oriented Approach” in the specialty 123 "Computer Engineering" "Bachelor" education degree contains 85 pages of explanatory notes, 13 figures, 8 tables, 2 appendixes. The volume of graphic material is 2 sheets of A3 format.

The aim of the qualification work is to develop a software for synthesis of actors models within the subject-oriented approach.

The task of data organization in a distributed environment is considered. To solve such tasks, there are methods, algorithms and software based on parallel calculations. Most of the tools analyzed require complex computational procedures and often lose performance as data increases.

Models for the implementation of distributed systems, which are grouped on separate objects - actors, are studied. Such models are quite fast in implementation, and their operation is based on sending messages between actors. Each actor is able to send messages, process them and create new actors. The software for implementation of actor models are considered and the Akka framework is noted.

The synthesis algorithm of actor models based on the Akka framework and the visual approach are developed, which takes into account all the properties of models, is characterized by speed and simplicity.

The distributed system based on actor model for organization of data with shared memory is built. Its software implementation is carried out and its efficiency is confirmed by the results of experiments.

The software system for synthesis of actor models based on the visual approach contains an additional level of abstractions and is quite effective in the implementation of parallel and distributed computing.

Keywords: ACTOR MODEL, SUBJECT-ORIENTED APPROACH, DISTRIBUTED SYSTEM, AKKA FRAMEWORK.

ТЕХНІЧНЕ ЗАВДАННЯ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

1.1 Програмний засіб синтезу моделей акторів на основі предметно-орієнтованого підходу.

1.2 Область застосування – розподілені обчислення.

2. ОСНОВА ДЛЯ РОЗРОБКИ

Основою для розробки є завдання на кваліфікаційну роботу, затверджене кафедрою комп'ютерної інженерії факультету комп'ютерних інформаційних технологій Західноукраїнського національного університету.

3. ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою кваліфікаційної роботи є розробка автоматизованої системи керування температурою води у тепломережах на основі нечіткої логіки.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелами даної розробки є матеріали навчальної і реферативної літератури, технічна документація, науково-дослідні роботи, журнали.

5. ТЕХНІЧНІ ВИМОГИ

5.1 Основні функціональні вимоги до комп'ютерної програми:

5.1.1 універсальність – кожен програмний модуль програми може бути довільно замінений на модуль з подібними функціональними можливостями;

5.1.2 простота удосконалення – в структурі системи повинні бути передбачені принципи горизонтального (збільшення функціональних можливостей окремих блоків) так і вертикального (збільшення кількості функціональних блоків) нарощування;

5.1.3 стійкість до збоїв;

5.2 Вимоги до надійності:

5.2.1 розроблене програмне забезпечення повине бути надійним, що забезпечить роботу системи при будь-яких умовах.

5.3 Умови експлуатації:

5.3.1 наявність встановленої операційної системи Windows;

5.3.2 наявність встановленого пакету приладних програм Matlab;

5.3.3 код програмних модулів повинен містити необхідні для його розуміння коментарі.

6. ВИМОГИ ТЕХНІКО-ЕКОНОМІЧНОГО РОЗДІЛУ

6.1 Розрахунок техніко-економічних показників розробки проекту.

6.2 Розрахунок ринкової ціни розробленого програмного забезпечення.

6.3 Обґрунтування економічних переваг розробленого програмного продукту у порівнянні з аналогом.

7. ПОРЯДОК КОНТРОЛЮ

7.1 Представлення кваліфікаційної роботи на попередній захист.

7.2 Представлення кваліфікаційної роботи на захист.

Завдання прийняв до виконання _____ Складанюк В.Є.
(підпис) П.І.П. студента

Керівник кваліфікаційної роботи _____ Савка Н.Я.
(підпис) П.І.П. керівника

ЗМІСТ

					КР.КІ.07171/19.00.00.000 ПЗ		
Змн.	Лист	№ докум.	Підпис	Дата			
Розробив		Складанюк В.М.			ПРОГРАМНИЙ ЗАСІБ СИНТЕЗУ МОДЕЛЕЙ АКТОРІВ НА ОСНОВІ ПРЕДМЕТНО-ОРІЄНТОВАНОГО ПІДХОДУ	Літ.	Арк. 8 / Акрушів 85
Перевір.		Савка Н.Я.					
Консульт.		Савка Н.Я.				ЗУНУ.ФКІТ. КІ-42	
Н. Контр.		Мельник Г.М.					
Затвердив		Березький О.М.					

ВСТУП

Надійність програмного забезпечення стає все більш актуальною задачею, оскільки зараз практично не залишилося сфер людської діяльності, не схильних до комп'ютеризації та інформатизації. Помилки в програмах, а також недоступність будь-яких інформаційних сервісів здатні призвести до суттєвих збитків. Надійність програмного забезпечення включає два аспекти: здатність обробляти велику кількість запитів та відновлюватися після збоїв.

Для вирішення цих задач, зазвичай, використовують ту чи іншу форму паралельної обробки даних. У першому випадку паралельна обробка дозволить обробляти більший об’єм даних чи більшу кількість запитів одночасно. У другому випадку дозволить продовжити роботу системи в цілому, якщо якась її частина вийшла з ладу.

На сьогоднішній день існує дві моделі паралельних обчислень: модель з пам'яттю (shared memory) і модель з посилкою повідомлень (message passing) або модель актора. У більшості сучасних мов програмування реалізовано підтримку першої моделі, що включає механізми розробки процесів і потоків та інструменти, що забезпечують спільний доступ до розділеної пам'яті (механізм блокувань).

Незважаючи на широке поширення, даний підхід має чимало суттєвих недоліків. По-перше, паралельні програми, які використовують модель із пам'яттю, дуже складно писати і налагоджувати. Зазвичай, це пов'язано з тим, що такий підхід є не природним, він практично не зустрічається у реальному житті. Складнощі при налагодженні програм виникають, оскільки стан гонки (race condition) практично унеможливлює точне відтворення умов виникнення помилки. Взаємоблокування часто є помилкою, налагодження якої є досить трудомістким процесом.

По-друге, такі програми мають обмежену здатність до масштабування. В той же час виникає проблема, коли необхідно забезпечити розподілену роботу програми для підвищення продуктивності і забезпечення відмовостійкості.

Для подолання бар’єру складності проектування та керування реалізацією складної розподіленої системи створено підхід із використанням окремих функціональних сутностей – акторів. Модель актора позбавлена вищевказаних недоліків, оскільки в ній відсутня колективна пам'ять. Для обчислень використовують актори, які взаємодіють один з одним на основі надсилання повідомлень. Кожен актор у відповідь на повідомлення може приймати локальні рішення, створювати інших акторів, посилати нові повідомлення або прийняти рішення про те, як реагувати на прийняте повідомлення.

Найдосконалішим на даний момент засобом для розгортання подібних систем, побудованих на базі моделі акторів є програмний інструментарій Akka. Проте навіть він має певні недоліки. Опис взаємодії та поведінки акторів, а також їх конфігурація переважно знаходяться в одному місці та є досить низькорівневим. Як наслідок, для систем з багатьма акторами і, відповідно, сценаріями спілкування між ними текст опису є громіздким та неочевидним.

Підтримка такого файлу конфігурації є доволі складною. Розширення подібних систем з точки зору розгортання є досить витратним процесом щодо ресурсів та часу. Окрім того, актори не мають вбудованого механізму композиції, тому присутнє обмеження можливості статичної типізації повідомлень, які опрацьовуються.

Для подолання вищезазначених недоліків розроблено концепцію Visual Akka, яка за допомогою комбінації предметно-орієнтованого та візуального підходів потенційно спрощує роботу програмістів та системних аналітиків при побудові акторних систем.

Саме подолання вищеописаного недоліку і розробку відповідного інструментарію ставить собі у мету дана робота.

Зважаючи на вищезазначене, метою кваліфкаційної роботи є розробка моделей акторів на основі системи Visual Akka, що забезпечить для спрощення побудови та реалізації розподілених систем, із використаням інструменту компонування аторів. Для досягнення мети необхідно виконати такі завдання:

– дослідити особливості моделей акторів;

– проаналізувати програмні засоби реалізації акторних моделей;

– розробити алгоритм синтезу моделей акторів на основі візуального підходу;

– розробити програмний модуль реалізації моделей акторів;

– на основі проведених експериментів дослідити ефективність розробленої акторної моделі;

– здійснити техніко-економічні розрахунки показників ефективності розробки проекту.

Об'єкт дослідження. Розподілені процеси при опрацюванні великих масивів даних.

Предмет дослідження. Алгоритм синтезу моделей акторів на основі предметно-орієнованого підходу.

Практичне значення отриманих результатів. Програмний засіб синтезу моделей акторів із застосуваням предметно-орієнтованого а візуального підходів.

Основні результати дослідження опубліковано на V науково-практичній конференції «Інтелектуальні комп’ютерні системи та мережі» [28]. Копії публікації наведено у додатку А.

1 ПРЕДМЕТНО-ОРІЄНТОВАНИЙ ПІДХІД ДО МОДЕЛЮВАННЯ СИСТЕМ

1.1 Аналіз предметно-орієнтованого підходу

Термін предметно-орієнтована мова (Domain Specific Language, DSL) позначає мову програмування або моделювання, що застосовується для вирішення конкретного кола задач в термінах, максимально наближених до певної предметної області [8]. Ось деякі приклади предметно-орієнтованого них мов. Мова SQL, що використовується для роботи з базами даних і є одним з найбільш, напевно, відомих і успішних предметно-орієнтованих мов. Макромова пакета Excel, призначена для табличних обчислень. Make – мова і утиліта для автоматизації збірки програмного забезпечення головним чином для Unix платформ. За останні п'ять років спостерігається сплеск появи нових мов програмування – Ruby, Groovy, Scala, Lua, Nemerle і ін.

Разом із збільшення потреб розробників ще однією причиною цього сплеску є значний прогрес продуктивності обчислювальних засобів, що дозволяє легко реалізовувати компілятори і середовища розробки, дотримуючись класичного компіляторного методу [13, 29]. Всі підходи до розробки DSL можна розділити на дві категорії:

1) компиляторні підходи, що дозволяють реалізовувати предметно-орієнтовану мову як окрему та самостійну;

2) підходи, в яких нова мова виходить розширенням базових.

Технології розробки компіляторів поділяють на дві групи – ті, які засновані на атрибутних граматиках, і ті, що базуються на принципі переписування [13, 31]. Різні атрибутні граматики і системи переписування лежать в основі всіх традиційних методів розробки компіляторів. В межах компиляторного підходу існують такі технології [13, 36]:

1) засновані на атрібутних граматиках – Elegant, FNC-2, CDL;

2) засновані на системах переписування – Term Processor Kimwitu, TXL, ASF + SDF.

В межах підходів, в яких нова мова виходить розширенням базової, існують метод макророзширення (на прикладі реалізації системи для тестування мережевих протоколів) і метод синтаксичних розширень (на прикладі технології Camlp4) [13, 29].

Розглядаючи особливості реалізації програмних мов на основі предметно-орієнтованого підходу, оцінюють їх, виходячи з врахування нижче перелічених критеріїв.

1. Семантична замкнутість. Підхід не відповідає даним критерієм, якщо з його допомогою реалізуються DSL, в яких присутня велика кількість конструкцій, які не мають семантичного трактування в термінах, відповідних до поточної предметної області.

2. Трудомісткість реалізації. Підхід дозволяє реалізовувати DSL без надмірних витрат. В іншому випадку ідея створення і реалізації нової DSL може виявитися невиправданою в порівнянні із використанням вже готових універсальних мов.

3. Трудомісткість внесення змін до специфікації і реалізації DSL. Підхід не відповідає такому критерію, якщо невелика зміна в предметної області тягне трудомістку переробку специфікацій DSL і / або створених за допомогою такого підходу коштів підтримки DSL. Критерій найбільш актуальний для тих DSL, предметні області яких активно розвиваються і видозмінюються із плином часу.

4. Оптимальність реалізації. Дуже важливо, на скільки підхід дозволяє створювати оптимальні реалізації DSL, так як часто це є одним з основних чинників при ухваленні рішення про створення та реалізації нової DSL.

5. Підтримка оптимізації на рівні предметної області. Ця оптимізація дозволяє часто суттєво збільшити ефективність коду, оскільки використовує знання про особливості предметної області.

6. Збереження «прив'язки» до тексту особливо актуально при розробці DSL за допомогою розширення базової мови, по скільки дозволяє реалізувати налагодження, видачу повідомлення про помилки компіляції в термінах DSL, а не в малоінформативних термінах базової мови.

Існує чимало методів розробки компіляторів, які можуть бути застосовані до реалізації DSL. Як правило, компиляторний підход використовується для реалізації тих предметно-орієнтованих мов, які сильно відрізняються від існуючих. Наприклад, випадок, коли треба лише додати кілька конструкцій в якусь вже наявну мову програмування, обходяться простішими засобами.

Основні парадигми, що використовуються при розробці компіляторів – це атрибутна граматика (attribute grammars) і переписування (tree / term rewriting).

Атрибутна граматика введена Дональдом Кнутом в кінці 60-х років, як продовження ідеї обчислення семантики цілого виразу через семантику його підвиразів. У результаті появився підхід до побудови різноманітних автоматичних засобів обробки мов на основі їх формальних специфікацій із подальшою автоматичною генерацією компіляторів, відладчиків і т. д.

Атрибутна граматика – це контекстно-вільна граматика з атрибутами, які визначають семантику конструкцій мови. Семантика програми визначається набором значень атрибутів кореневого вузла. Атрибути бувають успадкованими і тими, що синтезуються. Успадковані атрибути використовуються для передачі семантичної інформації вниз по дереву розбору програми, а синтезовані – вгору. Приклади синтезованих атрибутів – довжина спискуі сума його елементів, а приклад успадкованого атрибута – середнє значення всіх елементів списку.

З теоретичної точки зору успадковані атрибути зайві, але на практиці вони застосовуються досить часто, особливо при використанні значення атрибута поза контекстом, в якому він був створений. Найбільше поширення успадковані атрибути отримали при реалізації багатопрохідних компіляторів.

Інші приклади атрибутів для опису семантики мови програмування: code (синтезується код цільової платформи у вигляді інструкцій у формі «код інструкції», «аргумент»), name (синтезується ім'я константи), value (синтезується значення константи або число), envs (синтезується оточення, тобто таблиця символів з полями «ім'я», «значення»), envi (успадковане оточення) та ін. Очевидно, що за допомогою подібних атрибутів можливо задати компілятор мови програмування.

Кнут сформулював таку умову, що гарантує можливість розв'язання атрибутної граматики. Якщо побудувати граф залежностей атрибутів (атрибут «a» в ньому має дугу в вершину, позначену атрибутом «b» тоді і тільки тоді, коли значення атрибута «b» залежить від значення атрибута «a»), то він повинен бути ациклічним, тобто на ньому можна визначити частковий порядок обчислень. Ця умова є досить суворою і важко реалізованою на практиці. Тому існує багато інших, більш слабких умов, заснованих на тому факті, що на практиці важливо не так виконати всі можливі обчислення, скільки обчислити необхідний набір атрибутів в корені дерева.

Існують різні методи, що спрощують опис обчислень в атрибутних граматиках. За довгу історію свого розвитку атрибутна граматика реалізована різними програмними засобами. Крім синтезованих і успадкованих атрибутів, введених Дональдом Кнутом, з'явилися також атрибути-колекції (collections), залежні від усього дерева розбору, рекурсивні атрибути (circular attributes) і атрибути-посилання (referenced attributes).

Різні системи, засновані на атрибутних граматиках, дозволяють використовувати мови реалізації, такі як C (Elegant, FNC-2), LISP (FNC-2), ML (FNC-2), Java (Jast-Add), Scala (Kiama).

Основними недоліками компіляторного підходу є:

1) компілятори, автоматично створені за допомогою атрибутних граматик, часто виявляються неефективними щодо часу виконання і використання пам'яті;

2) у атрибутних граматиках часто відсутні зручні модульні засоби. Таким чином, специфікації мов програмування, розроблені на основі атрибутних граматик, часто виявляються надзвичайно громіздкими, зокрема в них важко вносити зміни. Цей є серйозною перешкодою при розробці DSL, оскільки специфікації таких мов схильні часто змінюватися.

3) обчислення в атрибутних граматиках грунтуються на незмінності вхідного дерева. Ця властивість забороняє будь-які оптимізації обчислень, що вимагають перетворення вихідного дерева;

4) обчислення в атрибутних граматиках є досить жорсткими – наприклад, не можна обчислити значення атрибута, якщо не враховано всі значення атрибутів, від яких він залежить. Щоб не захаращувати такими залежностями семантику, доводиться створювати штучні конструкції.

Існує велика кількість методів і логічних формалізмів, що використовують процедури послідовної заміни частин формул або термів формальної специфікації відповідно за певними правилами, які називаються правилами переписування (Rewriting Rules) [13, 18]. Вперше правила переписування були введені в лямбда-численні А. Черчем. Найпростішим прикладом переписування в графі є заміна при оптимізації в дереві програми множення на два складанням.

Вносити невеликі зміни в DSL, описану за допомогою будь-якої системи переписування, простіше, ніж в атрибутних граматиквх. У підходах, що базуються на переписуваннях, найгострішим питанням є наявність середовища для повного циклу реалізації DSL, як правило, такий не пропонується.

Під багатьма мовами програмування, наприклад C / C ++, LISP, вбудовані макромови, які дозволяють прикладним програмістам в текстах своїх програм здійснювати заміну спеціально оформлених фрагментів програми на інший текст відповідно до деякимх заданих шаблонів. Для цього в тексті програми повинні бути описані як самі шаблони, так і випадки їх використання, що виконується за допомогою спеціальних конструкцій макромови (так званих макроконструкції).

З формальної точки зору макромова дозволяє задати систему переписування, а макропроцесор, який запускається до компілятора, виконує задані правила переписування і таким чином позбавляє текст програми від макроконструкції, є універсальним інструментом переписування, який, однак, використовується не розробниками компіляторів, а звичайними прикладними програмістами. Фактично за допомогою макроконструкції можна ввести в мову програмування нову синтаксичну конструкцію. Саме ця можливість робить макромови зручним і природним способом розробки невеликих DSL на основі універсальних мов програмування.

При всій простоті реалізації DSL методом макророзширенні від має суттєві недоліки:

1) відсутність різних синтаксичних перевірок для нових конструкцій, наприклад, перевірок відповідності типів параметрів;

2) при перекладі препроцесором конструкцій нової DSL в конструкції базової мови втрачається їх «прив'язка» до початкового тексту програми, що ускладнює розуміння програмістами повідомлень про помилки компіляції і суттєво ускладнює покрокове налагодження програми. При цьому програмістам доводиться розбиратися в тому коді, який макроконструкції зручно приховували при розробці програм.

3) ніхто не заважає користувачеві ввести свої макроси, можливо перевизначивши при цьому вже наявні, що призводить до серйозних помилок в роботі програми;

4) користувач так реалізованої DSL ніяк не обмежений в можливостях базової мови і, отже, може створювати програми, що виходять за межі даної предметної області, тобто у даному підході відсутня семантична замкнутість.

Метод синтаксичних розширень (syntax extensions) заснований на ідеї запровадження в існуючу мову нових конструкцій і семантичних правил їх трансляції у базову мову. Проекції нових конструкцій в базову мову задаються на основі правил синтаксичних розширень. Програма, що оперує іншою програмою, наприклад, її породжу – ідея, яка лежить в основі всіх синтаксичних розширень. Причому, на відміну від макророзширення, при визначенні нових конструкцій нам доступна інформація про граматику базової мови.

1.2 Аналіз програмних засобів на основі предметно-орієнтовного підходу

Ідея генератора компіляторів, побудованого на основі атрибутних граматик, з'явилася в дослідницькій лабораторії компанії Philips в 1987 році. При цьому

передбачалося поєднати в одному підході атрибутні граматики і «ледачі» обчислення (lazy evaluations). Перша версія системи Elegant вийшла в 1992 році, як внутрішня розробка компанії Philips. До 1997 року роботи по Elegant підійшли до завершення і компанія випустила цей продукт як вільно розповсюджуваний [30, 43].

В системі Elegant існує власна мову для специфікації компіляторів. Систем є імперативною і включає об'єктно-орієнтовані і функціональні риси. У мові Elegant є поліморфізм, функції вищих порядків, частково застосовні функції, зіставлення за шаблоном (pattern match- ing).

Кожен тип в Elegant є множина (навіть примітивні типи, такі як Int). У мові підтримується також концепція лінійного успадкування типів (тобто забороняється успадкування від двох і більше типів), є узагальнені функції (gener- ics) і можливість «перевантаження» функцій. Специфікації, створені за допомогою цієї мови, система Elegant компілює в ANSI C. Для опису правил атрибутної граматики в Elegant використовуються часткові функції, які дозволяють задавати набір різних правил в залежності від різних вхідних значень, що містять їх функції [43].

Хоча система Elegant і являє собою інструмент для підтримки повного циклу розробки компілятора, але він ніяк не усуває зазначених вище недоліків атрибутних граматик.

Система FNC-2 створена у французькому дослідницькому інституті INRIA. Розробка почалася в 1986 році, перший прототип вийшов в 1989 році. Після цього система FNC-2 використовувалася для розробки декількох великих додатків компілятора для паралельної логічної мови Parlog в код абстрактної машини SPM, компілятора з мови ISO-Pascal в P-code, транслятора мови E-LOTOS [17, 34]. Система FNC-2 також використовувалася для власної компіляції в процесі розробки.

При створенні системи FNC-2 особливо досліджувалися можливості підвищеня ефективності обчислень у атрибутних граматиках, оскільки головна робота всієї системи відбувається саме при обчисленні атрибутів. Тут важливо

поставити ефективний порядок обчислень, в зв'язку з чим запропоновано три типи обчислювачів:

1) вичерпний (exhaustive);
2) інкрементальний (incremental);
3) паралельний (parallel).

Вичерпний обчислювач є найпростішим. Принцип його роботи заснований на тому, що обчислювач мінімальний час витрачає на вибудовування порядку обчислень, використовуючи порядок, заданий ззовні. Цей обчислювач може почати свою роботу в будь-якому місці дерева атрибутної граматики.

Наступний тип обчислювача – інкрементальний, дозволяє виконувати семантичні функції контролю, призначення яких визначати ситуації, коли один і той же атрибут обчислюється кілька разів, і позбавлятися від зайвих обчислень.

Паралельний обчислювач призначається для прискорення процесу обчислення атрибутів за рахунок паралельної роботи. Таким чином, користувач FNC-2 може побудувати найбільш ефективне обчислення атрибутів в своїй системі, комбінуючи наявні типи обчислювачів.

Крім обчислювачів, FNC-2 містить й інші функціональні блоки для підтримки процесу реалізації компілятора: генератори лексичних / синтаксичних аналізаторів, генератори абстрактного атрибутного дерева, засоби для модульного опису всієї системи і взаємозв'язків таких модулів. Модульність є однією із основних переваг системи. FNC-2 дозволяє розбити описувану атрибутну граматику на модулі, для котрих обчислення атрибутів можуть бути проведені незалежно один від одного.

Таким чином стає можливим виробляти оптимізації в проміжках між роботою різних обчислювачів, в різних частинах системи, а також модифікувати атрибутне дерево в ці моменти часу.

Для задання атрибутної граматики в системі FNC-2 використовують мову OLGA [13, 34]. Ця мова підтримує типи-дерева, модульну структуру і є строго типізованою. У OLGA є поліморфізм для множин списків, а також «перевантаження» функцій, присутні конструкції для задання виключень і

зіставлень за шаблоном. Нововведенням є атрибутні класи, що дозволяють задавати правила для цілої групи атрибутів одночасно, що значно скорочує розмір специфікацій. У FNC-2 існує кілька генераторів коду – в C, LISP, ML, C / fSDL.

Найбільш зрілими є генератори в мови C і ML. Можливість вибору з двох мов, що відрізняються парадигмою програмування (імперативна, в разі C, і функціональна, в разі ML) є суттєвою превагою системи. Проте із ростом предметних областей зростає потреба в предметно-орієнтованих мовах, що відрізняються за своїм властивостями від більшості наявних мов програмування, зокрема від C і ML. Тому використання тільки цих двох парадигм не вирішує проблеми створення різноманітних за своїми властивостями DSL.

Система CDL. Якщо в описі граматики задіяний нетермінал, все входження якого перетворюються однаково, то такий нетермінал називається афіксом (affix), а відповідна грамматака – афіксальною. Compiler Description Language (CDL) є програмною системою і мовою для розробки компіляторів на основі афіксальних граматик. Ця система була вперше представлена в 1971 році і грунтувалася на досить незвичайній концепції, а саме в CDL були відсутні базові примітиви для найпростіших операцій (наприклад, арифметичних операцій), і ці операції доводилось реалізовувати самостійно.

Важливим кроком стало додавання засобів розробки модулів, що дало можливість використання CDL2 у великих проектах Версія системи CDL3 з'явилася в 2004 році і використовується в основному тільки в академічних цілях, оскільки практично не має документації. До того ж вона видає неінформативні повідомлення про помилки, що суттєво ускладнює налагодження коду. Однак, на відміну від попередніх версій, CDL3 має реалізацію базових примітивів і систему типів.

Для створення DSL засобами CDL потрібно описати грамматику і правила обчислення афіксів. Правила записуються на мові CDL, яка спеціально орієнтована на конструкції-дерева. У мові існує чотири стандартні операції для роботи з такими структурами:

- об'єднання (join);

- розщеплення (split);
- рівність (equal);
- присвоювання (assign).

Щоб побудувати нове дерево із двох інших за допомогою деякої операції OP, треба всього лиш написти:

[EXPR1 OP EXPR2 -> EXPR].

Для того, щоб перевірити, що дерево зібрано з двох інших дерев за допомогою операції OP, досить написати правило:

[EXPR -> EXPR1 OP EXPR2].

У правій частині правила дозволяється використовувати код на цільовій мові (найчастіше це код на мові Асемблер) і таким чином задавати спосіб обчислення афіксів. Основною проблемою використання CDL є те, що мова не підтримує іменування змінних, тобто змінні одного типу називаються за допомогою послідовних номерів (наприклад, EXPR1, EXPR2 і т.д.). Мова не підтримує також операції логічного заперечення, тому потік управління базується на успіху або невдачі правил оброблюваної граматики. Проте відсутність зрозумілої документації для останньої версії CDL3 ускладнюють використання мови для реалізації предметно-орієнтованих мов.

Проект Term Processor Generator Kimwitu є спільним дослідницьким проєктом кількох університетів Королівства Нідерландів, які орієнтовані на створення системи для розробки програм, які працюють з деревами або термами, зокрема компіляторів. Система Kimwitu приймає на вхід абстрактний опис термів із директивами для реалізації і опис функцій на цих термах, а на виході видає код на мові ANSI З, що визначає структури даних для термів і функції роботи з цими термами (створення термів, виділення пам'яті і т .п.), а також реалізацію правил переписування [47].

Як і багато інших подібних інструментів, Kimwitu вперше був випробуваний для генерації власного коду. Надалі за допомогою цієї системи розроблялися інші компілятори, а також різноманітні інструменти тестування. Одним з найвідоміших застосувань Kimwitu було його використання при реалізації мови LOTOS1 [36, 39]. В цьому проекті за допомогою Kimwitu були згенеровані структури даних і методи введення / виведення.

Основним засобом специфікації в Kimwitu є алгебра термів, що описує базові терми і операції над ними. Зовнішні і внутрішні описи структур в системі однакові, тому можна легко розбити опис термів і функцій на кілька модулів і працювати з ними окремо, об'єднуючи в потрібні моменти результати роботи. Кожен тип терма відповідає якомусь типу в мові С, тому при перевірці типів Kimwitu покладається на перевірку типів в С-компіляторі.

Реалізація DSL за допомогою цієї системи є сладною, оскільки від розробника нової мови потрібно створити безліч функцій на мові С, що оперують складними структурами даних над термами. При цьому можливості мови в існуючій версії порівняно невеликі.

У 1985 році дослідниками з університету міста Торонто була запропонована система (мова і програмний засіб) TXL, призначена для вирішення задач на основі переписуваня і заснована на концепції функціональних мов і парадигмі продукційних правил (rule-based paradigm) [14, 36]. Будь-яка програма на мові TXL складається з двох частин:

- опис вихідних структур за допомогою граматик в нормальній формі Бекуса-Наура (BNF);
- набір правил для переписування термів.

На вхід у програмі будуються деревовидні структури, які потім перетворюються згідно із поданими на вхід правилам перепсування, а із отриманого перетвореного дерева будується вихідна специфікація. Таким чином, можна виробляти «переписування» з однієї мови в іншу, тобто реалізовувати компілятором.

Правда, для цього доведеться описати граматику для вхідної мови, щоб перетворювати вхідний текст програм у вираз на основі дерева, а потім в підсумковий виконуваний код, а також правила і функції переписування. Система TXL застосовується для вирішення таких задач, як переклад програм з одних мов програмування на інші, перетворення різних формальних специфікацій, а також структурованих документів, трансформація схем баз даних, розпізнавання математичного рукописного тексту, аналіз мережевих пакетів та ін. Система використовується як в науковх та комерційних цілях.

Сам по собі базова мова TXL дуже проста і мало функціональна, тому описати на її основі будь-яку складну систему дуже непросто. Однак існують функціональніші діалекти-розширення TXL, наприклад, Evolving TXL, в якому додані такі можливості [31]:

- правила зіставлення за зразком із поверненням помилки часу виконання в разі невиконання жодної відповідності (Must Matching Rules);
- правила, які виконують не зіставлення за зразком або заміну, а засоби для розміщення блоків службового коду (наприклад, для визначення змінних, опису допоміжних процедур);
- сильна типізація, призначена для того, щоб задавати унікальний тип, до якого може бути застосоване те чи інше правило;
- вихідні параметри для можливості повертати значення з правил;
- оператор розгалуження (відома конструкція if-then-else);
- параметризація правил типом даних;
- модульність.

Система ASF + SDF представляє середовище розробки і на-бір інструментів для реалізації нових мов программірованя, для генерації і прототипування засобів підтримки розробки ПЗ, а також для створення аналізаторів коду і проведення різного роду перетворень над кодом. Перші версії системи з'явилися ще на початку 90-х років минулого століття. Для роботи з системою ASF + SDF слугує середовище розробки MetaStudio IDE, яке включає компілятори для двох

основних мов – ASF і SDF, генератори парсеров, бібліотеку ATerms і кілька візуалізаторів різних проміжних рішень [31].

Syntax Definition Formalism (SDF) є мовою для опису контекстно-вільних граматик. ЇЇ відмінною особливістю є можливість модульного опису синтаксису мов программуваня.

Algebraic Specification Formalism (ASF) є мовою для опису семантики нової мови програмування, граматика якої визначена за допомогою SDF. Семантика описується за допомогою правил переписування. Загальна схема роботи системи ASF + SDF виглядає таким чином:

- синтаксичний аналіз, тобто розбір граматики, описаної на SDF, і побудова по ній абстрактного синтаксичного дерева;
- препроцесування – виконання перетворень над тим абстрактним поданням, яке було отримано на попередній стадії (наприклад, зв'язування змінних в умовах, введення конструкцій if-then-else для правил з одинаково лівими частинами);
- генерація коду. На цій стадії для правил переписування породжується код на мові C. Вибір мови C обумовлений переносимістю і можливостями щодо оптимізації одержуваного коду;
- постпроцесування – виконання різних покращуючих претворень над отриманим.

Система ASF + SDF є універсальною і призначена для реалізації найрізноманітніших предметно-орієнтованих мов. Це позначається на розмірі специфікацій, які потібно створити при описі DSL, а також призводить до громіздкості і надмірності одержуваного коду. Також недоліком є той факт, що мова C – це єдина вихідна мова системи.

1.3 Характеристика методів організації даних розподіленому середовищі

В останнє десятиліття відбувається широкомасштабне впровадження в практику розрахункових робіт паралельно обчислювальної техніки (особливо кластерних архітектур), що сприяє відродженню інтересу до пакетної проблематиці, що вимагає реконструкції старих і створення нових методів і засобів організації паралельних і розподілених обчислень.

Якщо перші пакети прикладних програм (ППП) були простими тематичними добірки програм для вирішення окремих задач в тій чи іншій предметної області, то сучасні ППП є складними програмними комплексами, що включають: прикладне програмне забезпечення (функціональне наповнення) пакета у вигляді бібліотеки прикладних програмних модулів; системне програмне забезпечення для організації вирішення користувальницьких задач з використанням функціонального наповнення, що включає засоби автоматизації планування обчислень (в тому числі паралельних і розподілених); спеціалізовані мовні засоби опису предметної області і постановки призначених для користувача задач.

Початкова орієнтація і налаштування на широкий клас задач предметної області, в якій працює прикладний фахівець, є однією з ключових особливостей ППП. Поєднання в одному пакеті множини різноманітних моделей і алгоритмів досягається шляхом використання принципу модульної організації функціонального наповнення пакета.

Модуль представлено, як правило, у вигляді автономної програмної одиниці, написаної на традиційній мові програмування і забезпечує рішення деякої підзадачі. Передбачається, що взаємодія модулів здійснюється тільки на рівні їх вхідних і вихідних параметрів. Використання принципу модульності дозволяє замінити написання програми (в традиційному розумінні) її конструюванням з готових програмних блоків великого розміру. Розширення

класу задач, що вирішуються пакетом, може досягатися за рахунок підключення до ППП новостворюваних модулів.

В даний час серйозну увагу приділяється проблемі створення інтелектуальна них ППП, що дозволяють кінцевому користувачу формулювати свою задачу в змістовних термінах без вказівки алгоритму її вирішення. Синтез схеми рішення і складання цільової програми відбуваються автоматично, при цьому деталі обчислень залишаються прихованими від користувача.

Такий спосіб організації обчислень відомий як концептуальне (структурний, складальне) програмування. Він припускає наявність обчислювальної моделі (схеми) предметної області, що дозволяє описувати обчислювальні можливості ППП для вирішення задач певного класу. Таку обчислювальну модель можна визначити як сукупність значущих величин (параметрів) предметної області і функціональних відносин між ними. Таким чином, обчислювальна модель, по суті, визначає правила застосування і поєднання модулів в процесі виконання задачі і дозволяє автоматично здійснювати планування обчислень.

Інструментальний комплекс (ІК) ORLANDO орієнтований на створення ППП для однорідних UNIX-кластерів з можливістю автоматичної генерації паралельних програм для вирішення обчислювальних задач у вигляді композиції готових функціональних блоків [1, 4]. Метод ORLANDO надає ресурси для опису об'єктів предметної області (параметрів і операцій), відносин між ними і забезпечує на основі цих описів постановку обчислювальних задач в непроцедурному вигляді. Конструкції методу носять виключно описовий (декларативний) характер і не передбачають вказівки порядку виконання модулів, які беруть участь у вирішенні задачі.

Всі інформаційно-логічні зв'язки між модулями виявляються на етапі трансляції опису предметної області. Текстовий редактор надає користувачеві набір функцій, необхідних для швидкого і зручного формування опису предметної області і постановок задач із використанням мовних конструкцій ORLANDO, а також функцій для взаємодії з іншими компонентами

інструментального комплексу – транслятором, підсистемами компіляції і запуску, базою розрахункових даних.

Синтаксичний аналізатор здійснює розбір опису предметної області, перевірку коректності та цілісності такого опису і його переклад у внутрішнє представлення системи планування. Планувальник, реалізований відповідно до розглянутої вище базової моделі планування, будує по не процедурної постановки задачі з урахуванням наявних обмежень паралельний (в загальному випадку) план її вирішення у вигляді частково-упорядкованого набору імен обчислювальних модулів з функціонального наповнення пакета.

Порядок роботи методу ORLANDO передбачає, що на початковому етапі користувач формує опис предметної області, використовуючи виражальні засоби. Далі він приступає до рішення задачі, формулюючи її в непроцедурній формі "вихідні параметри => мета розрахунку" у вигляді вказівки планувальнику. Побудований планувальником паралельний план вирішення задачі на етапі трансляції перетворюється в паралельну програму на мові С ++, після чого готова паралельна програма переміщується на цільовий обчислювальний кластер, де компілюється штатними засобами. За допомогою підсистеми запуску з бази розрахункових даних на кластер копіюються необхідні результативніні дані, і здійснюється запуск програми.

Інструментальний комплекс DISCOMP призначений для підтримки основних етапів розробки і застосування розподілених ППП, орієнтованих на роботу в багатоплатформному розподіленомуобчислювальному середовищі, яке може включати обчислювальні кластери різних конфігурацій. Обчислювальні модулі, складові функціонального наповнення розподіленого ППП, – виконувані програми, які можуть бути реализовані на різних мовах програмування і бути платформо залежними.

Допускається включення до складу функціонального наповнення таких пакетів програмних комплексів, строго розміщених в певних вузлах, а також успадкованого программного забезпечення, що перестав відповідати сучасним вимогам, але до сих пір експлуатується у зв'язку із трудомісткістю його

модифікації або заміни. Віддалений запуск модулів, обмін даними між модулями через файли і моніторинг вузлів реалізуються засобами системної частини.

До основних складових програмно-апаратної архітектури інструментального комплексу DISCOMP відносяться: система управління РВС, набір обчислювальних клієнтів РВС, система зберігання даних і засоби доступу користувачів до ППП. Система управління РВС (серверна частина DISCOMP) підтримує взаємодію з підсистемами зберігання даних і доступу користувачів до пакету, а також забезпечує централізоване управління вузлами РВС. Серверна частина включає системне ядро, менеджери обчислювальних процесів і ресурсів, диспечер черги задач, підсистему журналізації і виконавчу підсистему.

Ефективність вирішення задач на основі заначеного методу забезпечується особливостями роботи DISCOMP: можливістю максимально ефективного використання різнорідних ресурсів РВС; підтримкою високопродуктивної взаємодії між розподіленими компонентами; гнучкою диспетчеризацією черг задач; наявністю ресурсів оптимізації обсягів даних, що передаються між модулями в процесі вирішення задач; можливістю динамічного управління процесами вирішення задач.

У сучасному світі все ширшу роль відіграють технології, що забезпечують ефективну обробку великих масивів даних. Сучасні програмні засоби пред'являють серйозні вимоги до обчислювальних ресурсів. Особливе значення приділяється алгоритмам і методам, що застосовуються для обробки і аналізу даних із використанням комп'ютерних кластерів. Однак реалізація процедур обробки даних на кластерних системах пов'язана з вирішенням таких задач, як змінити та розподіл даних між процесорами, балансування навантаження, обробка відмов, збір і агрегація проміжних результатів [4].

На даний момент виділяють дві базові моделі паралельних і розподілених обчислень [2, 4, 9]:

- модель обчислень в загальній пам'яті;
- модель обчислень в середовищі з передачею повідомлень.

Для моделі із загальною пам'яттю характерно, що всі обробники мають доступ до загальної пам'яті, і програміст вибирає, що буде виконувати кожен обробник. Системи із загальною пам'яттю можуть бути розширені до розподілених, якщо операційна система буде інкапсулювати спілкування між вузлами і забезпечить загальний доступ до пам'яті між усіма системами в кластері.

При використанні моделі паралельних обчислень із передачею повідомлень програміст вибирає структури мережі обчислювачів і програму, виконувану кожним обробником.

Одним із ефективних методів обробки великих обсягів даних в розподілених середовищаxx є метод MapReduce, запропонований компанією Google на початку 2000-х для сканування і обробки великої кількості сторінок з мережі Інтернет. Така модель відрізняється простотою і зручністю використання, приховує від користувача деталі організації обчислень на кластерній системі [10, 45].

Перевага MapReduce полягає у тому, що вона дозволяє розподілено виконувати операції попередньої обробки і згортки. Операції попередньої обробки працюють незалежно один від одного і можуть проводитися паралельно. Аналогічним чином робочі вузли здійснюють операцію згортки – для цього необхідно, щоб всі результати попередньої обробки із одним конкретним значенням ключа оброблялися одним робочим вузлом в один момент часу [45].

Паралелізм також дає деякі можливості відновлення після часткових збоїв серверів: якщо в робочому вузлі, що виконує операцію попередньої обробки або згортки, виникає збій, то його робота може бути передана іншому робочому вузлу (за умови, що вхідні дані для проведеної операції доступні). Користувачу досить описати процедуру обробки даних у вигляді декількох функцій, після чого система автоматично розподіляє обчислення по кластеру, обробляє відмови машин, балансує навантаження і координує взаємодії між машинами.

В межах концепції MapReduce припускають, що дані організовані у вигляді деякого набору впорядкованих записів, а їх опрацювання відбувається в три

стадії: Map, Shuffle і Reduce (див рис. 1.1). На стадії Map виконується попередня обробка і фільтрація данних за допомогою функції Map, яку виділяє користувач.

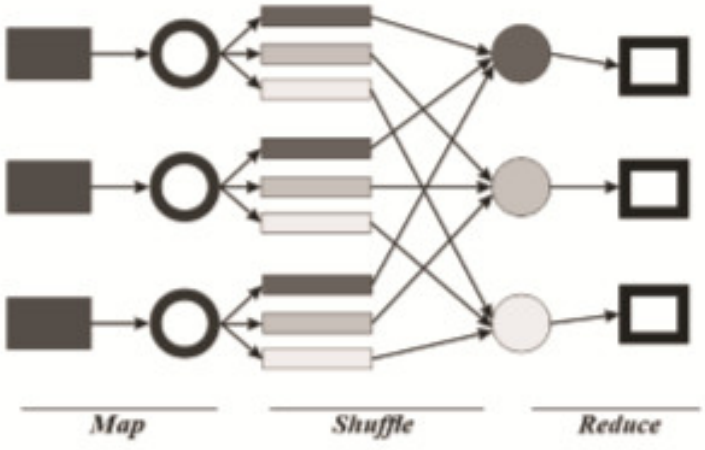

Рисунок 1.1 – Стадії методу MapReduce

Всі запуски функції працюють незалежно і можуть працювати паралельно, в тому числі на різних машинах кластера. Функція Map, як правило, застосовується на тій же машині, на якій зберігаються дані. Це дозволяє знизити передачу даних по мережі (принцип локальності даних) [45].

На стадії Shuffle висновок функції Map розбивається на спеціальні секції. Коженна секція відповідає одному ключу виведення стадії Map. Крім того, вона приймає на вхід сукупність записів, які відповідають даним ключу, і загальну кількість Reduce-завдань, а значенням, що повертається, є номер задачі, в якій опрацьовувався кожен запис. Кожна секція формується на основі функції хешування, яка викликається для кожного ключа і залежить від певних критеріїв, наприклад від номера задачі. Для прискорення процесу опрацювання інформації дуже часто на даній стадії застосовують алгоритми паралельного сортування.

Стадія Reduce. Кожна секція зі значеннями, сформована на стадії Shuffle, по падає на вхід функції Reduce. Ця функція обчислює фінальний результат для кожної окремої секції. Всі запуски Reduce, як і функція Map, працюють незалежно і можуть працювати паралельно, в тому числі на різних машинах кластера. Для

деяких видів обробки згортки не потрібно, і каркас повертає в цьому випадку набір відсортованих пар, отриманих базовими обробниками.

Парадигма MapReduce достатньо гнучка і може легко адаптуватися під різні типи завдач, включаючи додаткові стадії обробки інформації. В даний час широко використовується не тільки для ефективної обробки великих масивів даних, але і для вирішення прикладних задач, пов'язаних із розширеною обробкою тексту, сортуванням даних, індексуванням документів, статистичними аналізом, машинним навчанням, обробкою зображень.

Зважаючи на суттєві переваги описаного методу, він втрачає свою ефективність при вирішенні задач, у яких існує зв’язок між вхідними даними. За таких умов альтернативою виступає модель акторів, особливості якої розглянемо у наступному розділі.

1.4 Постановка задачі кваліфікаційної роботи

Описаний предметно-орієнтований підхід до розв’язування задач у розподіленому середовищі спрямований на деталізацію конкретної предметної області. Програмні засоби в межах цього підходу характеризуються суттєвими недоліками, зокрема, нзькою типізацією даних, відсутність системи компонування, низька продуктивність за рахунок неузгодженості потоків паралельних обчислень.

Методи паралельних обчислень у розподілених середовищах включають медоди із спільною пам’яттю та методи на основі передачі повідомлень. Вони здатні опрацьовувати великі масиви даних, включають інструментальні засоби граматики та типізації даних проте доссить часто є трудомісткими при реалізації. В той же час існуюча модель MapReduce досить гнучкою та здатною до адаптації при будь-яких нових класах задач. Метод володіє високим ступенем паралелізму, оскільки опрацювання даних розділено на три стадії, проте при розв’язуванні

задач із зв’язаними вхідними даними суттєво програє у продуктивності обчислень.

За таких умов виникає необхідність застосування таких методів, зокрема, акторних моделей, які б уможливлювали розв’язок задач із спільною пам’яттю із належним рівним паралелізму. Тому метою кваліфікаційної роботи є розробка програмного засобу синтезу моделей акторів в межах предметно-орієнтованого підходу.

Для розв’язку поставленої задачі у роботі необхідно виконати такі задачі:

– дослідити особливості моделей акторів;

– проаналізувати програмні засоби реалізації акторних моделей;

– розробити алгоритм синтезу моделей акторів на основі візуального підходу;

– розробити програмний модуль реалізації моделей акторів;

– на основі проведених експериментів дослідити ефективність розробленої акторної моделі;

– здійснити техніко-економічні розрахунки показників ефективності розробки проекту.

2 АЛГОРИТМ СИНТЕЗУ МОДЕЛЕЙ АКТОРІВ

2.1 Особливості моделей акорів

В даний час в розробці багатопотокових додатків домінує підхід використання загального змінного стану – великої кількості об'єктів, кожен з яких працює в своєму потоці. Об'єкти характеризуются станом і можуть бути змінені в будь-який момент в різних частинах програми. Зазвичай, в такому коді для синхронізації взаємодії потоків накопичується багато блоків, керованих замками (примітивами синхронізації). Останні використовуються для забезпечення контрольованої зміни стану і запобігання ситуації, коли дкілка потоків вирішують змінити стан одночасно. Проте надмірна кількість таких блоків може суттєво знизити швидкодію програми, оскільки потоки довго чекатимуть вивільнення необхідних ресурсів.

Головною проблемою використання низькорівневих конструкцій синхронізації (замки, потоки) є складність визначення поведінки і роботи програми. У таких системах часто з'являються дефекти («боротьба за ресурси), які можуть бути не знайдені на етапі тестування, але можуть призвести до суттєвих помилок при завантаженні і використанні програм на реальних серверах.

Для вирішення вищезгаданих проблем використовують модель акторів, яка дозволяє писати ефективні багатопоткові програми і легко аналізувати поведінку програми під час роботи. Поштовхом до розробки моделі стала необхідність підтримки паралельних обчислювальних систем, які містять велику кількість незалежних процесорів, кожен з власною локальною пам'яттю і процесом обміну повідомленнями, і спілкуються через швидкісні мережі передачі даних» [11, 22, 35].

Перші статті про моделі акторів появилися на початку 70-их років 20 століття і були спрямовані на дослідження штучного інтелекту. Оригінальна стаття К. Хьюїта і Г. Бейкера вийшла ще у 1977 році [35]. Проте зовсім скоро акцент застосування моделі змістився в бік розподілених обчислень. Тоді ще не

існували MPI, не було ні справжніх, дійсно розподілених систем і навіть сама мережа в її сучасному вигляді тільки починла з'являтися. Однак, це не завадило обговорювати всікі теоретичні способи створення систем, здатних обробляти дані послідовно.

Суть моделі акторів дуже проста – розподілені обчислення потрібно виконувати в середовищі гомогенних незалежних вузлів, кожен з яких або може виконати операцію сам, або переслати її іншому. Операції об'єднані у «завдання», які мають унікальний заголовок. Мета – адрес актора і повідомлення, яке актор прочитає із свого поштового ящика, коли задача туди потрапить. При цьому, повідомлення може містити «завдання» для інших акторів, які в ході виконання будуть «розіслані».

Таким чиом, ідея моделі акторів полягає у тому, що в ній всім є актор – і вузли, і повідомлення, що передаються між ними. Такий підхід дозволяє реалізувати так званий виклик-на-замовлення, використовуючи «повідомлення» і їх вміст в якості «обіцянки» (promise). Обіцянку буде виконано тим актором, якому було передано повідомлення із завданням, коли він його обробить.

Особливості моделі акорів:

- гомогенність обчислювальних вузлів;
- асинхронне виконання завдань;
- оригінальний розподіл акторів по вузлах.

Внутрішню структуру актора представлено на рисунку 2.1. Актор являться сукупністю черги, яка називається поштовою скринькою, конвеєра, який займається обробкою повідомлення в поштовій скриньці, «адреси» та стану. Оскільки у кожного актора є умовний адрес, то між собою вони можуть обмінюватися повідомленнями безпосередньо. Конвеєр задач виконує задачі з поштової черги і змінює стан всього актора.

Актор – це обчислювальна сутність, яка у відповідь на повідомлення може паралельно:

- послати кінцеву кількість повідомлень іншим акторам;
- створити кінцеву кількість інших акторів;

- визначити поведінку у відповідь на наступне повідомлення.

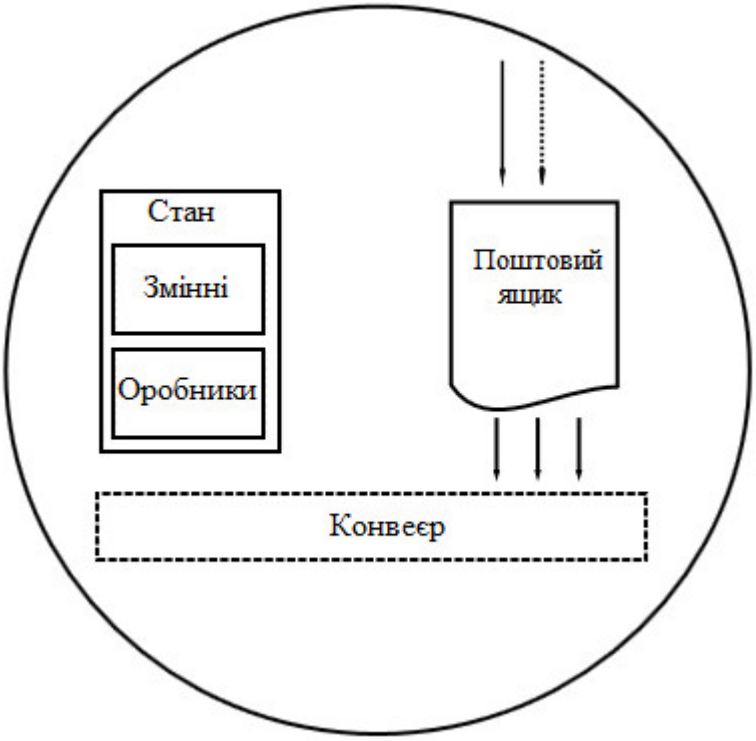

Рисунок 2.1 – Внутрішня стуктура актора

Актори взаємодіють між собою асинхронно, тобто актор, що послав повідомлення, не чекає відповіді від адресата, а продовжує своє виконання. Одержувачі повідомлень ідентифікуються за допомогою адреси, тому актор може посилати повідомлення тільки тим акторам, адреси яких він знає [35].

Розглянемо, яким чином модель актора дозволяє забезпечити надійність і відмовостійкість програм. Одним з основних переваг цієї моделі є хороша здатність до масштабування Для обчислень використовуються незалежні актори, що працюють паралельно. Для вирішення складної задачі можна створити більше акторів, і, відповідно до закону Густафсона, вирішити велику задачу за той же час, що і меншу, за рахунок паралельної обробки.

Не можоиво виконати те ж саме, використовуючи модель обчислень із пам'яттю, оскільки велика кількість паралельних потоків буде намагатися отримати доступ до загальної пам'яті, створюючи тим самим великі черги, в яких потоки будуть простоювати, очікуючи доступу до даних. За рахунок

масштабування модель актора дозволяє обробляти більший обсяг даних і більшу кількість запитів, тим самим, підвищуючи надійність і доступність додатків.

Актори взаємодіють, посилаючи один одному повідомлення. Перевагою такого підходу є те, що процеси можуть розташовуватися як на одному комп'ютері, так і на різних. При цьому на рівні моделі не потрібно додаткових дій для взаємодії віддалених акторів, а відміну від моделі з пам'яттю. Потоки можуть мати доступ до загальної пам'яті тільки в разі роботи на одному вузлі. Якщо вони працюють на різних вузлах, необхідний механізм обміну даними між вузлами, що ускладнює як модель, так і реалізацію.

Тому посилка повідомлень, притаманна моделі актора, дозволяє вирішити проблему масштабування додатків, реалізованих на основі цієї моделі. У програмі, що складається з великої кількості акторів, простіше реалізувати відмовостійкість. Для цього необхідно, щоб одні актори здійснювали моніторинг інших. Для цього актор може періодично посилати іншому актору деяке повідомлення, на яке останній повинен відповісти. Якщо відповіді не отримано, значить, актор завершив свою роботу. Також актор перед аварійним завершенням може посилати інформацію про факт завершення і умовах, при яких це завершення було виконано, іншому актору, щоб той, у свою чергу, зробив необхідні заходи для виправлення ситуації, що склалася.

Наприклад, він може створити нового актора або перерозподілити навантаження між тими, що залишилися. Якщо актори будуть виконуватися на різних вузлах, то це дозволить забезпечити відмовостійку роботу системи навіть при фізичному виході з ладу частини з них.

Взаємодія акторів відбувається через обробку «задач». У кожному завданні є цільовий актор, щодо якого необхідно застосувати повідомлення. Верифікація мети відбувається за аким алгоритмом:

1) мета є раніше відомою для актора;

2) мета набуває розголосу в момент отримання актором повідомлення із завданням, що містить мету;

3) мета є адресою актора, який буде створений після опрацювання отриманого повідомлення.

Таким чином мережа адрес акторів є заздалегідь відомою та зв'язаню, а ситуація, коли актор отримує коректне повідомлення з невідомою адресою, неможлива.

Обчислення в такій моделі протікають за рахунок обробки повідомлень в «задчах» акторами. Тобто, процес обчислень є «керованим даними» (data-driven), а не процесами (process-driven). Кожен актор обробляє тільки ті задачі, які потрапили в його поштову скриньку, при цьому в процесі обробки йому також необхідно підтримувати «поведінку заміни».

У середині поштової черги повідомлення мають такий порядок, в якому вони увійшли. У деяких працях обговорють механізм імплементації умовно безмежної поштової черги, яка може опрацьовувати повідомлення, що надходять одночасно. Сьогодні така обробка повністю реалізується транспортними рівнями програмного забезпечення за допомогою буферазації.

Конвеєр завдань визначає поведінку актора. У загальному випадку воно являє собою функцію обробки вхідних повідомлень. Якщо актор приймає повідомлення і воно потрапляє на конвеєр виконання, то відбуваються послідовно такі дій:

- вказівник черги поміщається на поточне необроблене повідомленн;.
- далі, процес обробник:
  - створює замість попередньої функцію-обробник, щоб вона продовжила обробляти повідомлення поштової скриньки;
  - якщо є вказівки, створює завдання відповідно до вихідного повідомлення;
  - якщо є вказівки, створює нових акторів;
- процес-обробник завершує свою роботу, що функція заміни стає основною.

Життєвий цикл конвеєра задач представено на рисунку 2.2.

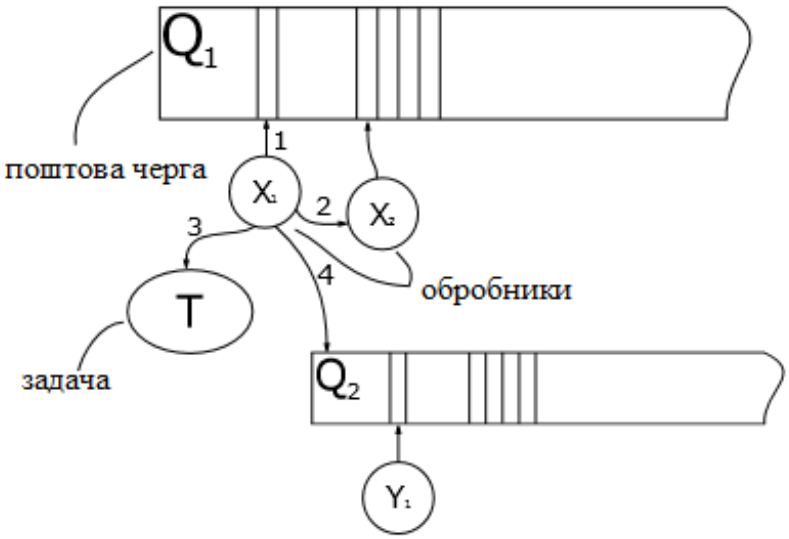

Рисунок 2.2 – Життєвй цикл актора

Володіючи таким життєвим циклом акторів, модель позиціонує як високо паралельна, з «успадковуючим» паралелізмом породжених акторів.

Розглянемо модель з точки зору надійності. У цй моделі не розглядається питання надійності акторів, надійності передачі повідомлень між ними, надійності результатів обчислень. Фактично, процес обчислень неможливо відновити лише при умові його повного перезапуску. Через те, що будь-який актор може взаємодіяти з будь-яким іншим, в разі помилки в одному з них, не можна буде визначити, від кого прийшла «завдача» і як ї відновити. У моделі відсутня ієрархія процесів, ієрархія акторів, немає ніяких зв'язків між породжуючими завдаами і задачами «батьками», хоч в реальних додатках подібний зв'язок завжди є. Можливо, це наслідки початків акторної моделі, проте застосування неієрархічної моделі до високопродуктивних обчислень призведе до проблем при збоях.

Введення ієрархії помітно спрощує процес розподілу ресурсів [26]. Як приклад, розглянемо випадок простого бінарного дерева – якщо один з фізичних вузлів породжує множину підзадач під час обчислень, то він може делегувати їх виконання своїм сусідам по дереву і далі до листя. Той же випадок, але без ієрархії призведе до помітних тимчасовим витрат при знаходженні найменш завантаженого вузла, тому що зведеться до повного перебору.

Основна ідея моделі акторів з одного боку досить проста, але в той же час фундаментальна. Явний поділ учасників процесу обчислень на незалежні вузли, явний опис процесу їх взаємодії і накладаючих на нього обмежень дозволяють перетворити майже будь-який алгоритм в набір задач і акторів. Модель акторів доцільно використовувати, як основу для нового механізму, який, не порушуючи базових принципів, уможливить збереження і відновлення стану елементів системи в будь-який момент часу після технічного збою.

2.2 Фрейворк Akka як програмний засіб реалізації акторів

Мова акторів (Actor language) – гнучка і потужна, що дозволяє створювати програми, окремі модулі яких можуть бути як сильно, так і слабо пов'язані [35]. До недавного часу існувало дві найбільш використовувані реалізації моделі акторів – бібліотека Scala.lang.actors, що надходила разом зі стандартним середовищем розробки для Scala, і частина бібліотеки Akka, присвячена саме реалізації моделі акторів. Обидві бібліотеки розглядали реалізацію моделі і відповідні допоміжні компоненти, як частина Scala, а саме як конкретні класи, що реалізують відповідні інтерфейси і підтримують поведінку, що відбиває властивості моделі з урахуванням внутрішніх особливостей реалізації.

На сього дняшній день існує безліч імплементацій базової моделі акторів у вигляді фреймворків для різних мов програмування. Одним з найпопулярніших є фреймворк Orbit для мови Java. Він реалізує актоів а межах JVM (JavaVirtualMachine), які в будь-який момент часу можуть бути активними чи не активними. Актори обмінюються повідомленнями асинхронно і разом складають розподілену систему. Включення і виключення актора залежить від вхідних повідомлень і націлене на економію ресурсів. Стан актора зберігається поза ним, наприклад, в базі даних. Фреймворк гарантує, що в системі одночасно може бути активна тільки одна копія актора, і що серненя до актора відбувається посідовно.

В якості основи для робти акторів Orbit пропонує середовище виконання «сцена» (stage), через яку розробнику стають доступні актори. Насправді сцена є ні що інше, як сервер для акторів-процесів. Передбачається, що на кожен вузол в кластері буде встановлена тільки одна «сцена». На відміну від вихідного опису моделі акторів, «задачі» в Orbit являються поверненням значення після передачі асинхронного повідомлення і виконання дій, тобто принцип «все є актор» тут в повній мірі не дотримується.

Мова Erlang – функціональна мова програмування. Вона розроблена в 1986 році в компанії Ericsson для створення додатків для телефонії. Предметна область наклала свій відбиток на Erlang. Телефонія за своєю природою паралельна – комутатори обробляють десятки і сотні тисяч одночасних з'єднань. Програми повинні забезпечувати відмовостійку роботу, оскільки вихід з ладу навіть на короткий час обладнання або програмного забезпечення може призвести до серйозних наслідків. В Erlang використовуються паралельні процеси для структурування програми. Вони не мають разподіленої пам'яті і взаємодіють один з одним за допомогою посилки асинхронних повідомлень. Процеси Erlang можуть створювати інші процеси. Процеси Erlang є «легкими» і належать мові, а не операційній системі, тому програма на Erlang може мати десятки і навіть сотні тисяч паралельних процесів [23].

Легко помітити, що процеси Erlang володіють подібними властивостями з акторами. Саме тому можна говорити, що Erlang є прикладом практичної реалізації моделі актора. Erlang володіє описаною вище здатністю до масштабування, що полягає в можливості створення великої кількості паралельних процесів, здатних ефективно виконуватися як на багатоядерному і обладнанні, так і в розподіленої середовищі.

Обмін повідомленнями між процесами, расподілено на різних вузлах так само просто, як і між процесами, що працюють на одному вузлі. Ця можливість вбудована в мову і практично не вимагає додаткових зусиль від програміста. В Erlang реалізована система супервізорів. Два процесу можуть бути пов'язані таким чином, що вони будуть спостерігати один за одним. Якщо один з пов'язаних

процесів завершиться, другому буде надіслано повідомлення про це із зазначенням причини.

Припустимо, один з зв'язаних процесів завершився через спроби поділу на нуль. Перед завершенням процес пошле повідомлення своєму супервізору про це із зазначенням конкретної причини. Супервизор на підставі цієї інформації може прийняти рішення про створення нового процесу або про завершення. Якщо буде прийнято рішення про створення нового процесу, то програма буде не тільки відмовостійкою, але і самовідновлювальною після збоїв [27]. Успішність ідей, реалізованих в Erlang, підтверджується реалізацією на практиці.

Розробниками мови Scala було прийнято рішення про припинення подальшого розвитку бібліотеки Scala.lang.actors і включеня саме реалізації на базі бібліотеки Akka в стандартну систему програмування для Scala. Тому в подальшому варто зосередитися на інструментарії, що надається розробнику фреймворком Akka [47].

Опис підходу до внутрішньої реалізації моделі акторів і обгрунтування обраних архітектурних рішень представлено в [38]. Сучасні рішення для створення паралельних програм і їх реалізації за допомогою Scala описано в [37]. Коли створюємо акторів засобами бібліотеки Akka, разом з ними під час виконання програми і на етапі компіляції створюються супутні компоненти, що забезпечують коректну роботу.

Основні елементи, які створює Akka:

- Actor (конкретний екземпляр актора);
- Mailbox (ящик повідомлень);
- Dispatcher (диспетчер);
- ActorRef (проксі над актором).

Структуру їх взаємодії передставлено на рисунку 2.3.

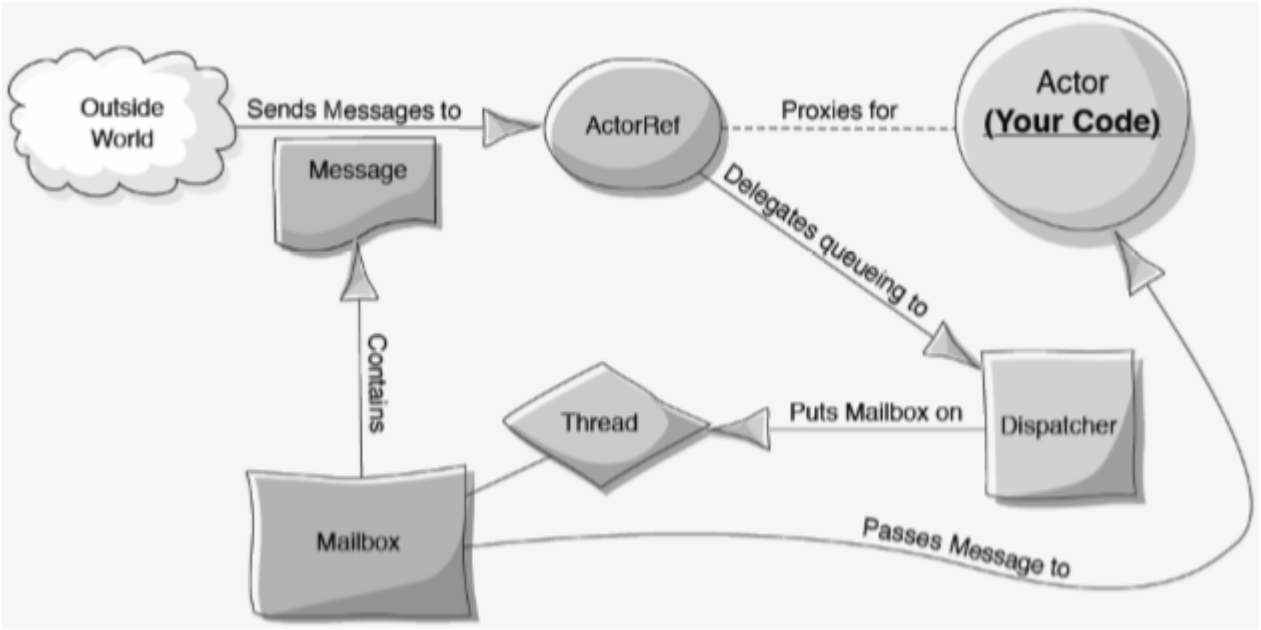

Рисунок 2.3 – Актор у Akka та його компоненти

Надсилаючи звіт про проблеми акторові, з ним безпосередньо ніколи взаємодія не відбувається. Замість цього відсилається повідомлення до об'єкта-проксі – ActorRef. Проксі звертається до диспетчера з приводу розміщення повідомлення в черзі повідомлень поштової скриньки актора. Диспетчер розмістить повідомлення в потік (thread), і коли той відпрацює, то дістане з черги одне із повідомлень і відправить його спеціально визначеним методом receive для обробки.

Варто пам'ятати, що як тільки повідомлення положено (доставлено) в ящик, той, хто здійснює виклик, може поступати, як забажає. Едине блокування – це розміщення повідомлення в чергу. Після цього, вся інша робота і обробка здійснюються в іншому потоці. Таке зменшення зв'язності системи дозволяє надати значну функціональність реалізації акторів в бібліотеці Akka.

Актори – «живі» об'єкти, що реагують на повідомлення, що надходять ззовні. Тому недоцільно моделювати за допомогою акторів щось статичне. Є тільки один спосіб звернутися до актора – відправити йому повідомлення. Аналогічно, єдиний спосіб щось зробити з цим повідомленням – опрацювати його в режимі receive.

Повідомлення на мові Scala реалізуються засобами особливої структури – часткових класів (case classes). Об'єкти таких класів – незмінні, значення їх атрибутів задаються в конструкторі при створенні. Також вони реалізують спільний з Java інтерфейс Serializable, що дозволяє вільно передавати об'єкти-сповідомлення по мережі.

Akka не допускає явного створення екземплярів акторів за допомогою оператора створення екземплярів об'єктів (в Java і Scala це оператор new). Для створення актора слід звернутися до екземпляра класу ActorSystem. Об'єк єкти цього класу служать «фабриками» для створення об'єктів акторів. Виклик відповідного методу класу ActorSystem повертає, як результат своєї роботи, не пряме посилання на об'єкт класу актора, а екземпляр класу ActorRef, за допомогою якого повинні відбуватися всі взаємодії з актором.

Даний підхід «непрямого звернення» до об'єкта забезпечує, перш за все, прозорість дислокації (location transparency), тобто реальне розташування примірника не має значення. Також виникає можливість реальним чином оптимізувати топологію програм під час виконання, змінюючи місцезнаходження актора. «Дотичне звернення» дозволяє використовувати модель «нехай впаде» («let it crash») для управління відмовами, яка уможливлює систему «вилікувати» себе, знищивши і створивши заново ті актори, які викликають помилки. Слід зазначити, що строкові ідентифікатори, що передаються в конструктор, і виклик методу actorOf досить важливі в Akka та уможливлюють надалі знаходити конкретні екземпляри акторів.

2.3 Алгоритм синтезу моделей акторів на основі візуального підходу

Алгоритм синтезу моделей акторів можна представити у вигляді мережі акторів, що взаємодіють один з одним. Мережу акторів можна представити у вигляді графа:

$$N = \{(a, m) | a \in A, m \in M\},$$

де a – актори,

m – повідомлення, що передаються від актора до актора.

Крім того, кожен актор може створювати і знищувати дочірні актори, які також можуть обмінюватися повідомленнями в мережі акторів. У повідомленні актор отримує всю необхідну для обчислень інформацію.В мережі акторів, що реалізує роботу алгоритму синтезу моделі акторів для інтелектуального аналізу даих, повідомлення можуть бути двох видів:

- сервісні повідомлення;
- обчислювальні повідомлення.

Сервісні повідомлення призначені для управління і налаштування функціонування мережі акторів. Сервісні повідомлення містять різну інформацію, за допомогою якої можна змінити налаштування актора або підмережі акторів. Наприклад, змінити кількість паралельно оброблюваних даних в підмережі паралельної обробки.

Обчислювальні повідомлення необхідні для передачі параметрів алгоритму між акторами. Обчислювальні повідомлення повинні містити:

- модель знань, яку будує алгоритм;
- дані, на основі яких перетворюється модель знань.

Модель знань містить якийсь стан роботи алгоритму, наприклад, будується дерево прийняття рішень для алгоритму побудови дерев рішень. Крім того, модель містить в настройки, на основі яких актори в мережі визначають, яким чином їм діяти в тій чи іншій ситуації, наприклад, максимальну або мінімальну кількість кластерів при кластеризації даних.

За допомогою даних, отриманих в повідомленні, актор здійснює операції, що перетворюють модель. Дані, які очікує актор, можуть бути будь-якого типу, але частіше за все для алгоритмів інтелектуального аналізу даних – це набори даних. Залежно від операцій, вироблених актором, це можуть бути також і набори

окремих значень з набору даних, наприклад, *i*-ті елементи, при розбитті на атрибутаи, будь-які інші дані, необхідні конкретному актору.

Варто зазначити, що в повідомленні, що пересилається, можуть міститися як безпосередньо дані і модель, так і посилання на них. Наприклад, на рисунку 2.4 зображено мережу акторів в разі загального джерела даних, до якого мають доступ усі актори в мережі. Суцільними стрілками позначена передача повідомлень між акторами, пунктирними – читання даних акторами з джерела даних. Очевидно, що передавати в повідомленні самі дані, а не посилання на них при такому джерелі даних недоцільно.

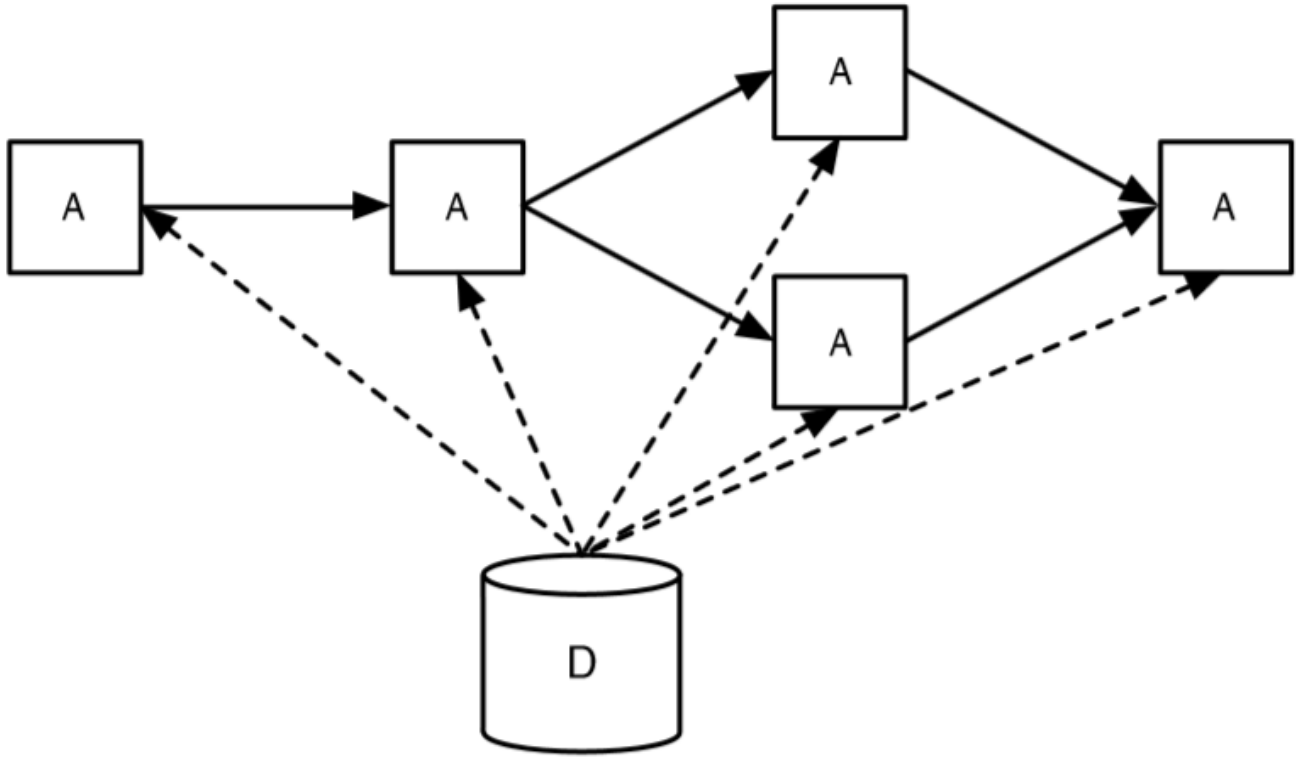

Рисунок 2.4 – Мережа акторів із спільною пам’яттю

У випадку, коли у кожного актора своє джерело даних з неідентичними наборами даних, є необхідність дані, які відсутні у актора, пересилати йому в явному вигляді. Зазначений випадок проілюстровано на КР.КІ.07171/17.00.00.000 С1.

Як приклад, можна привести систему збору та обробки даних із датчиків, де дані збираються з датчика в реальному часі і обробляються безпосередньо на

пристрої з таким датчиком, а потім передаються далі. Виходячи з цього, можна зробити висновок, що, в залежності від задачі мережі акторів і функціональності актора, існує один із способів передачі даних в повідомленнях між акторами: у вигляді посилання на дані в сховище або у вигляді самих даних. Крім того, найбільш ефективно буде мати у своєму розпорядженні актор в безпосередній близькості від джерела даних, з яким він взаємодіє.

Розглянемо окремий блок алгоритму, який є базовою одиницею для опису обчислень в алгоритмі. Блок є атомарним, тобто виконується як єдине ціле. Блок може бути представлений у вигляді окремого актора (див. рисунок 2.5), що приймає повідомлення з моделлю *m* і даними *d*, яка здійснює перетворення моделі і відправляє отриману модель знань *m** далі по мережі разом з даними. Стрілками на малюнку позначають шлях передачі повідомлення від актора до актора.

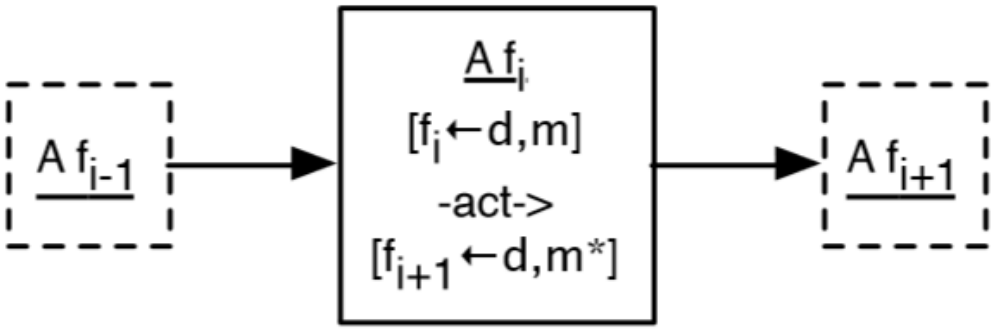

Рисунок 2.5 – Блок-схема окремої операції алгоритму

У мережі акторів послідовність представляється у вигляді ланцюжка акторів, які пересилають один одному повідомлення, що зображено на рисунку 2.6. Суцільними стрілками позначають шляхи передачі повідомлення між акторами всередині даної підмережі, пунктирними стрілками – повідомлення, що отримуються і передаються в зовнішнє середовище, щодо даної мережі. Кожен актор в підмережі приймає на вхід повідомлення з даними *d* і моделлю знань *m*, і здійснює над моделлю знань задані операції, причому перетворення може відбуватися, як з використанням переданих даних, так і без їх використання

тільки на основі отриманої моделі знань. Після завершення операції актор пересилає нову модифіковану модель знань і дані наступному актору.

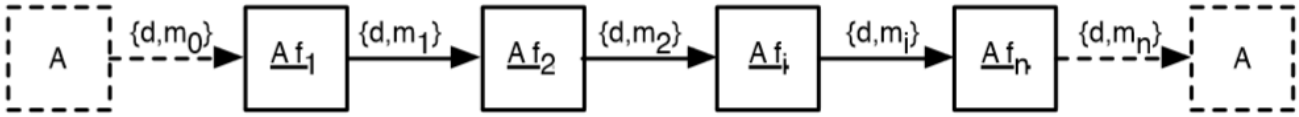

Рисунок 2.6 – Послідовність операцій

При отриманні повідомлення, що містить крім даних і моделі маркер reply-to і адресу актора, актор перевірить наявність наступного актора. Якщо у актора не вказано наступний актор, якому необхідно переслати повідомлення, то повідомлення буде відправлено актору вказаною у отриманому повідомленні.

Таким чином, ланцюжки акторів, складені за вищеописаною схемою, можуть бути перевикористані в декількох місцях мережі, і для їх використання необхідно крім даних і моделі передавати їм також маркер і адресу одержувача.

Одним з основних переваг алгоритму синтезу моделі акторів для інтелектуального аналізу даних у вигляді мережі акторів, є можливість їх паралельного виконання. Так як кожен актор працює незалежно (тобто інші актори не можуть безпосередньо втручатися в роботу актора і міняти його стан, а тільки через відправку йому повідомлень) і асинхронно від інших акторів в мережі, на основі мережі акторів можна будувати алгоритми для паралельної обробки даних.

Паралелізація за даними можлива в разі, коли алгоритм здійснює однакові обчислення над даними, тобто над кожним вектором даних здійснюються одні і ті ж операції, при цьому, якщо такі операції незалежні одна від одної, їх можна виконувати паралельно. У мережі акторів паралелізація за даними може бути реалізована кількома способами.

Оскільки найчастішим випадком для паралелізації обчислень за даними є циклічні операції, то паралелізацію циклічних обчислень в мережі акторів можна

представити, як мережу з актором маршрутизатором, ланцюжком акторів обчислювачів і актором агрегатором.

Актор маршрутизатор розбиває отриманий набір даних на набори меншого розміру за критерієм розбиття, дублює обчислювальний ланцюжок таким чином, що кількість отриманих ланцюжків буде дорівнює кількості наборів даних, і розсилає повідомлення всім ланцюжках. Блок-схем такого алгоритму наведено на КР.КІ.07171/17.00.00.000 С2. Після закінчення обчислень, ланцюжки акторів відправляють повідомлення з перетвореними моделями актору агрегатору, який об'єднує отримані моделі і передає їх далі по мережі.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ МОДЕЛЕЙ АКТОРІВ НА ОСНОВІ VISUAL AKKA

3.1 Структура програмного модуля синтезу моделей акторів

Для автоматизації процедури розробки моделі акторів на основі фреймворку Akka розроблено програмне забезпечення, діаграму використання якого проілюстрованго на рисунку 3.1.

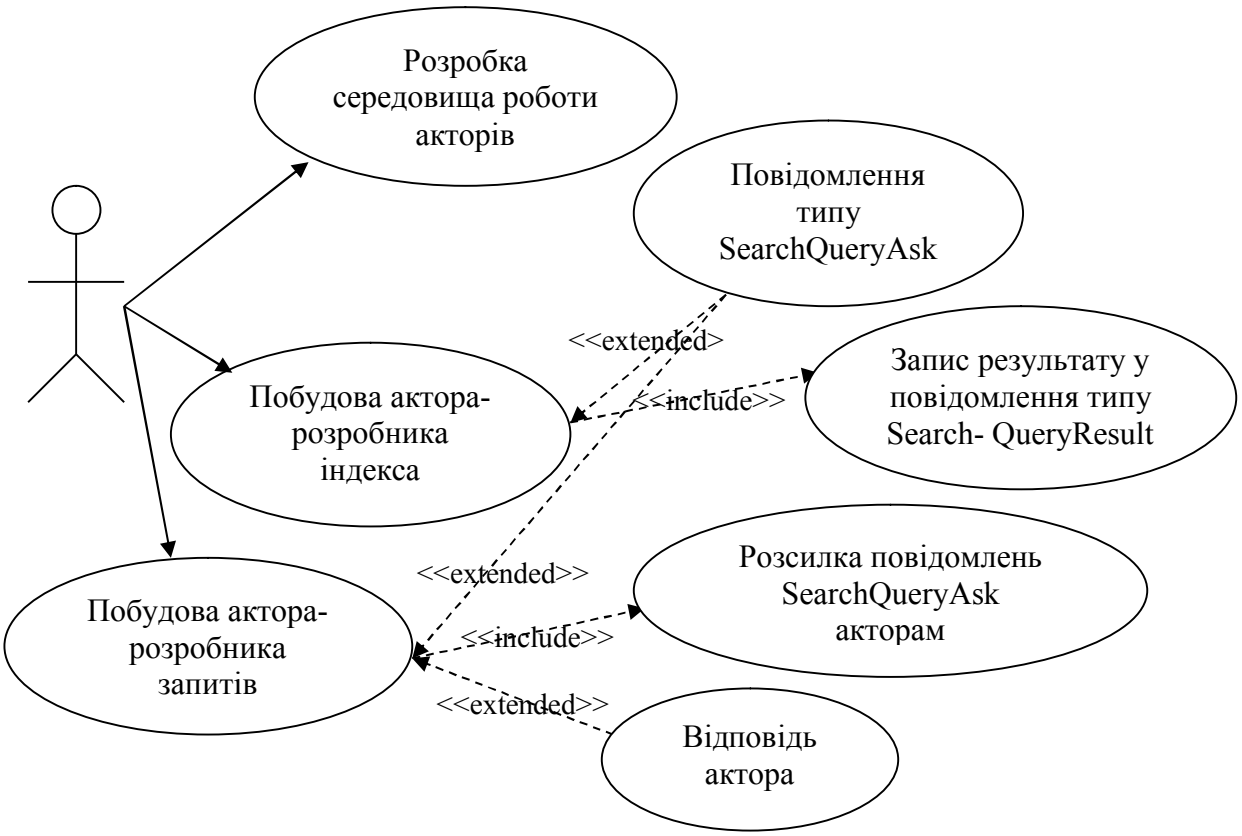

Рисунок 3.1 – Діаграма варіантів використання програмного модуля

Як видно і рисунка 3.1, програмна система включає функцію розробки середовища роботи акторів, функції розробки акторів, що індексують дані та акторів, що надсилають запити. У якості адміністратора системи може бути розробник моделі акторів або ж будь-який інший користувач з обмеженим доступом до функцій системи. Розглянемо детальніше особливості реалізації зазначних функцій.

Функція «Розробка середовища роботи акторів» відповідає за розроку середовища підтримки роботи та належного функціонування акторів. Система в будь-який момент часу може звернуся до середовища за допомогою IP-адрси, хоста та порта, тобто префікса фізичної адреси актора у середовищі.

Функція «Побудова актора-розробника індекса» включає підфункцію «Запис результату у повідомлення типу Search- QueryResult», а реалізація функції ґрунтується на повідомленні типу SearchQueryAsk, яке опрацьовується на основі методу public ArrayList <String> search (String word) класу PositionalIndex. У результаті отримуємо список імен файлів, що задовольняють вказаний запит, які записуються у повідомлення типу Search- QueryResult.

Функція «Побудова актора-розробника запитів» відповідає за розробку актора, який опрацьовує запити, та включає підфункцію «Розсилка повідомлень SearchQueryAsk акторам». На основі адрес акторів-індексаторів відбувається розсилка повідомлень та очікування відповіді не більше 5 с. Якщо відповіді немає від певного актора, то він вважається недійсним і запит у виляді повідомлення передається іншому актору.

В той же час, програмне забезпечення для синтезу моделей акторів має зручний користувацький інтерфейс, є надійним та швидким при виконанні будь-якої із вказаних функцій. Варто відзначити, що розроблене програмне забезпечення не вимагає ґрунтовних знань користувача із розробки програмних систем та навиків програмування. Йому достатньо розуміти особливості моделі на основі структурного елементу – актора, що потрібно задати на вхід моделі, щоб отримати бажаний вихід.

Реалізацію програмного модуля здійснено на основі фреймворку Akka та візуального підходу, тобто застосування інтегрованого середовища розробки програмнх засобів на основі графічних об’єків. Особливості реалізації розглянемо у наступному підрозділі.

3.2 Реалізація акорів для моделювання розподілених систем

Фреймворк для реалізації моделей акторів на основі фреймворку Akka та інструментів візуального прогрмування отримав назву – Visual Akka. Зазначений фреймворк містить додатковий рівень абстракції, досить постий у застосуванні та уможливлює реалізувати залежності між акторами та їх взаємодію.

Visual Akka включає [5]:

- візуальне програмування та конфігурування – розробка наочної та прозорої взаємодії між вузами (актоами), що написані на основі класичних мов прогпрамування;
- предметно-орієнтовану мова потоків, як проміжна між візуальним представленням та акторами, реалізованими на мові, наприклад, Java. Зважаючи на це, програмісти мають можливість розробляти та редагувати програмний код для опису акторів без додаткових програмних середовищ, а системні аналітики, у свою чергу, візуалізувати відредаговані отримані результати;
- платформу на основі Netbeans Platform, що уможливлює поєднання процедури рзрбки Java-коду із процедурою розміщення акторів на схемах Visual Akka.

Система складається із різних компонентів необхідних для створення кластерної мережі, її розгортання та моніторингу із використаням базових компонентів – акторів. На рисунку 3.2 представлено загальну архітектуру фреймворку.

Запити із мережі потрапляють до робочих вузлів, мережу яких побудовано за принципом децентралізованої мережі p2p. На кожному робочому вузлі також працює контролер розгортання Visual Akka, на основі якого можна автоматично розгорнути веб-рішення і стежити за його станом. У ролі робочого вузла виступає фізичний або ж віртуальний сервер чи обчислювальний пристрій, який підключений до мережі, що зв’язється із іншими вузлами.

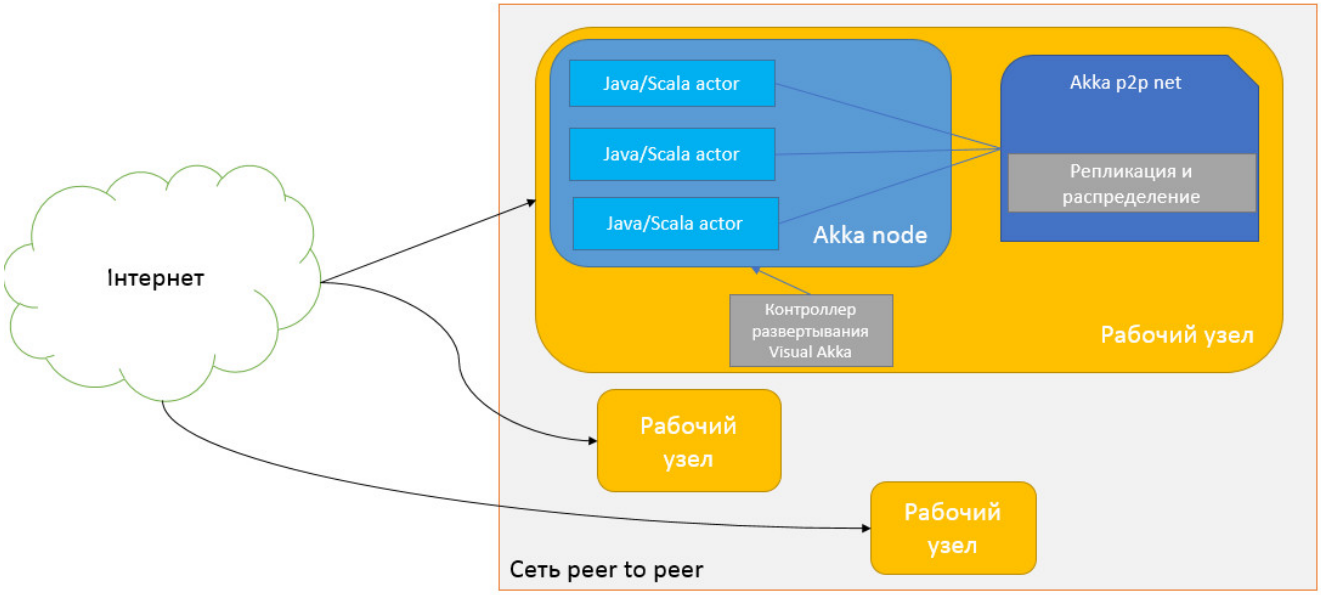

Рисунок 3.2 – Архітектура системи Visual Akka

Мережа p2p – це віртуальне об'єднання усіх вузлів у мережу типу pier-to-pier, що уможливлює синхронізацію загального стану виконання для усіх вузлів. Контролер розгортання – це спеціальний сервіс на сервері, що відповідає за автоматичне розгортання сервісу Visual Akka. Akka node – це сервер компонентів, на якому розгорнуті актори Akka. Akka p2p net підтримує стан мережі Akka та сповіщає про його зміну інші вули.

При запуску системи перше, що створюється – це середовище, в якому працюватимуть актори. На кожній машині створюється об'єкт ActorSystem, який буде підтримувати роботу з акторами. Всі інші актори будуть створені в цьому середовищі.

Після розробки середовища на будують два актори: актор-творець індексу, і актор- обробник запиту за загальним індексом. Перш, ніж перейти до опису акторів, слід визначити, якими повідомленнями вони можуть обмінюватися. В системі існує два типи повідомлень: SearchQueryResult і SearchQueryAsk. Повідомлення SearchQueryAsk направляється як запит від користувача до актора, який відповідає за індексування. Воно містить лише одне текстове поле query, конструктор класу і метод, за яким можна отримати запит public String getQuery.

Повідомлення SearchQueryResult містить масив із результуючими файлами, які вдповідають критеріям пошуку ArrayList <String> resultFileNames, конструктор і генератор результуючих імен файлів. Обидва повідомлення реалізують інтерфейс Serializ- able, оскільки для передачі повідомлень між акторами ці повідомлення необхідно серіалізувати.

Актор індексування називається Indexing- Actor. Він створює екземпляр класу Positional- Index, який і займається побудовою індексу. Файли, з яких буде розроблятися індекс, беруть із каталогу Books, який знаходиться в проекті. Актор індексування реагує на повідомлення SearchQueryAsk і обробляє їх. Всі інші повідомлення ігноруються. Після отримання повідомлення запит, який є у повідомленні, витягується і обробляється методом public ArrayList <String> search (String word) класу PositionalIndex. Метод приймає на вхід запит, а повертає список імен файлів, що задовольняють цей запит. Далі результат записується в повідомлення типу Search- QueryResult і надсилається відправнику. У результаті маємо такий метод обробки:

```
public void onReceive (Object message) throws Exception {
if (message instanceof SearchQuery.SearchQueryAsk) {
  String query = ((SearchQuery.SearchQueryAsk) mes- sage). GetQuery ();
  SearchQuery.SearchQueryResult resul = new Search-
Query.SearchQueryResult
  (this.index.search (query));
  getSender (). tell (result, getSelf ());
  } Else {
  unhandled (message);
  }
}.
```

Процес індексування і пошуку. Розглянемо підхід до індексування, а саме клас Posi- tionalIndex. Клас містить два поля: масив файлів ArrayList <File> files,

які потрібно проіндексувати, і результуючий індекс Map <String, TreeMap <Integer, TreeSet <Integer> >> dictionary.

Структура індексу така. Для кожного терміна, що зустрічається в тексті, зберігається список файлів, що містять цей термін, і для кожного файлу зберігається список-слово, позиції цього терміна. Розглянемо роботу методу індексування. Він в циклі зчитує по черзі всі файли зі списку, отриманого на вхід від актора, видаляє знаки пунктуації та зводить слова до нижнього регістру. Звичайно, потрібно використовувати додаткові методи обробки, такі як стемінг або лематизації.. Потім кожне слово обробляється окремо. Якщо вхідного слова ще немає в регістрі, то для нього створюються відповідні масиви і мережі, що додаються в індекс.

Якщо слово вже є в словнику, то, можливо, воно зустрічалося вже в цьому файлі або тільки в інших файлах. При обробці файлів ведеться лічильник індексів слів в тексті, який обнуляється перед розглядом наступного файлу. Після обробки всіх файлів повертаємо імена, відповідні вхідному запиту.

Розробка актора запиту. Актор, відповідальний за роботу із запитами, називається QueryActor. Його інтерфейс складається з поля для введення запиту, кнопки пошуку і поля для виведення результату. При натисканні на кнопку Search спрацьовує слухач (лісенер), передає запит на обробку акторові у вигляді повідомлення Search- QueryAsk. Актор запиту має список всіх акторів індексування у вигляді їх фізичної адреси. У разі розширення системи або перенесення її в іншу мережу, все, що необхідно змінити в програмі – це список адрес акторів:

private static final Map <String, List <String>> ACTORS = new HashMap <String, List <String>> ();

static { ACTORS.put ("A", new ArrayList <String> ()); ACTORS.get ("A"). Add ("akka.tcp :/ / IndexingSys- tem@192.168.0.100: 2552/user/indexActor");

ACTORS.put ("B", new ArrayList <String> ()); ACTORS.get ("B"). Add ("akka.tcp / / IndexingSys- tem@192.168.0.142: 2552/user/indexActor");

```
ACTORS.get ("C"). AddAll (Arrays.asList ("akka.tcp :/ / IndexingSystem@192.168.0.142: 2553/user/indexActor", "akka.tcp :/ / IndexingSystem@192.168.0.142: 2554 / user / indexActor "," akka.tcp :/ / IndexingSystem@192.168.0.142: 2555/user/indexActor "));
}.
```

Відзначимо, що всі актори розбиті на групи. Таким чином система підтримує реплікації. Коли актор отримує повідомлення від графічного інтерфейсу користувача, він розсилає кожному актору зі списку повідомлення SearchQueryAsk і чекає на відповідь:

```
ArrayList <String> totalResult = new ArrayList <String> ();
boolean success = false;
for (String group: ACTORS.keySet ()) {
for (String actor: ACTORS.get (group)) {
if (! success) {
Timeout timeout = new Timeout (Duration. Create (5, "seconds"));
Future <Object> future = Patterns.ask (get Con- text (). ActorSelection (actor), message, timeout);
try {
SearchQuery.SearchQueryResult result = (SearchQuery.SearchQueryResult) Await.result (future, timeout.duration ()); totalResult.addAll (result.getResultFile- Names ());
success = true;
} Catch (Exception e) {
System.out.println (actor + "is unreachable");
}
success = false;
}.
```

Якщо відповідь не приходить протягом п'яти секунд, то такий актор вважають недійсним для даного запиту, і запит перенаправляється до наступного актора з групи. Якщо жоден актор з групи не відповідає, то запит повертає значення з інших, і, незважаючи на це, виводить на екран:

gui.showResults (totalResult).

Розроблений підхід дозволяє включати в мережу будь-яку кількість машин, легко розширюється. Запити індексу можна спрямовувати з будь-якої машини в мережі. Побудова координатного індексу дозволяє обробляти фразові запити. Реалізована в системі підтримка робіт з реплікаціями забезпечує її стабільність.

3.3 Експериметальні дослідження моделі акторів для організації даних у розподілену середовищі

У акторній моделі актор – це блок, що реалізує деякі обчислення. Мережу таких акторів позначимо:

$$A = \{a_1, a_2, a_3, ..., a_p\}, \quad (3.1)$$

де a_p – актор, що обчислює (визначає індекс).

Таким чином, вираз (3.1) становитиме систему акторів.

Кожен обчислювальний актор виконує деяку функція f_i, на основі вибірки даних n :

$$m = a_p(f_i, n, m). \quad (3.2)$$

У підсумку, результатом роботи актора є змінена модель знань *m*, яку він передає іншому актору. Залежно від типу блоку алгоритму (обробляє або обчислювалює), що виконується актором, виділяють два типи акторів:

- обробники актори, які реалізовують обробні блоки і взаємодіють із даними: $m = a_p^n(f, n, m)$;
- обчислювачі актори, які виконують обчислювальні блоки і не взаємодіють із даними: $m = a_p^n(f, nil, m)$.

Мережу акторыв можна представити у вигляді множини моделей акторів, а також спеціальних акторів, що забезпечують роботу усієї мережі та входять до однієї із моделей:

$$A = \{r, p\} \cup A_1', A_2', ..., A_v', ..., A_w', \tag{3.3}$$

де *p* – актор, що виконує послідовну частину алгоритму і блок розпаралелювання paral;

r – спеціальний актор, диспетчер повідомлень.

У загальному випадку середовище, в якому виконується алгоритм синтезу моделей акторів для інтелектуального розподілу даних, складається із обчислювальних вузлів і вузлів, що зберігають дані:

$$R = H \cup S = \{h_1, h_2, h_3, ..., h_j\} \cup \{s_1, s_2, s_3, ..., s_k\}, \tag{3.4}$$

де h_j – вузол обчислення;

s_k – вузол сховища даних.

Зважаючи на це, модель акторів *C* у розподіленому просторі даних можна представити у вигляді:

$$C \to R \to A = C \to R \to (C \times R) = \left\{ \left(h_j, c_x^n\right), \left(s_k, c_y^a\right) \middle| c_x^n, c_y^a \in C, h_j, s_k \in R \right\}, \tag{3.5}$$

де A відображає результат відтворення моделі акторів C у розподіленому просторі даних.

На основі формального представлення моделі акторів, що формує систему акторів, можна розробити розподілену систему. Архітектуру зазначеної системи зображено на рисунку 3.3 із такими позначеннями:

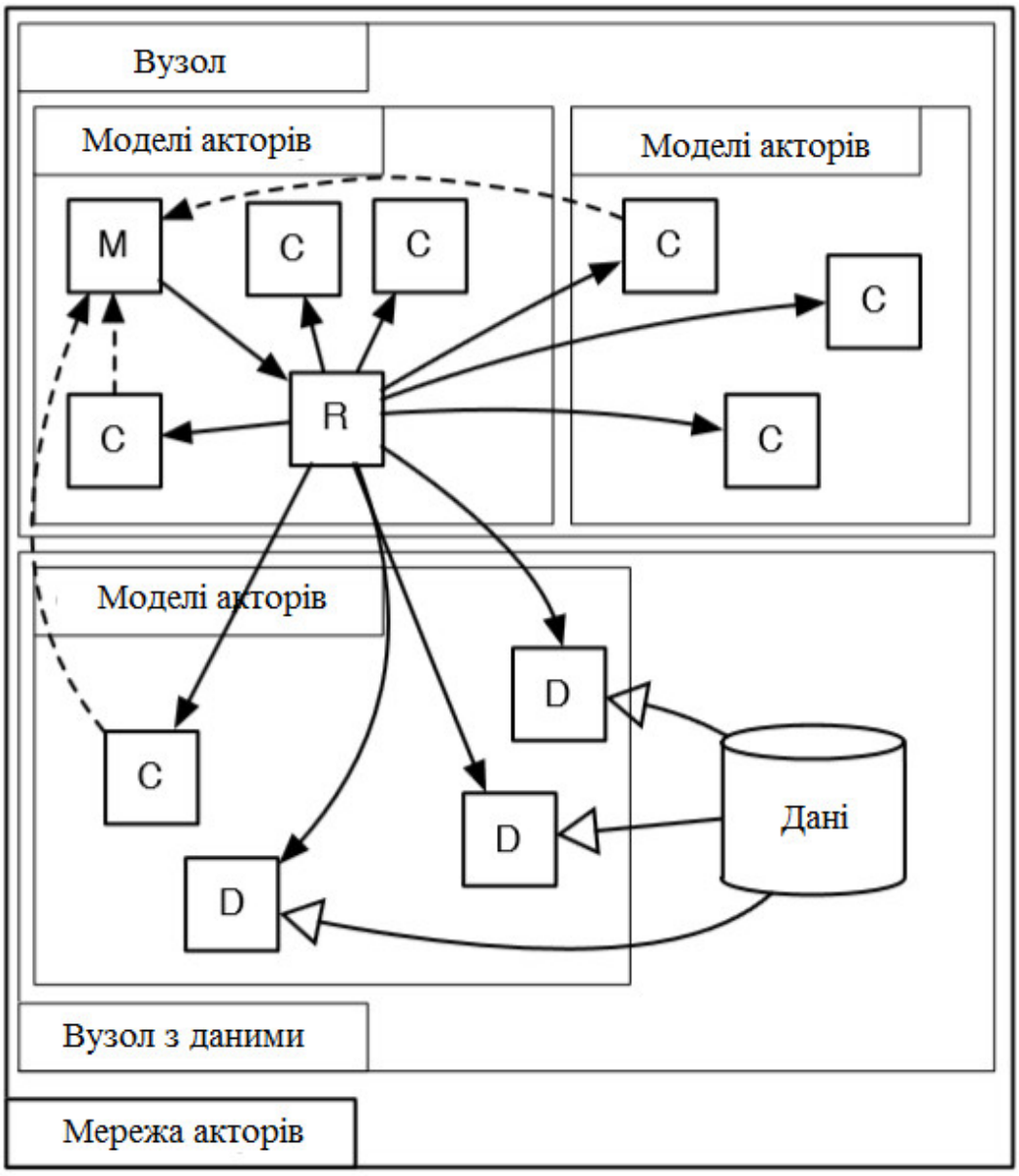

Рисунок 3.3 – Архітектура розподіленої системи на основі моделі акторів

На рисунку 3.3 використано такі позначення:

M – основний актор-обробник;

C – актори-обробники;

R – актор-маршрутизатор;

D – актори для роботи з даними.

Суцільною стрілкою позначено процес пересилання задач для обробки, а пунктирною стрілкою – перетворені моделі даних. Пустими стрілками позначено зчитування даних із сховища.

Розглянемо приклад реалізації моделей акторів для організації даних у розподіленому середовищі у системі Visual Akka Екранну форму плагіна для підтримки візуального Akka наведено на рисунку 3.4.

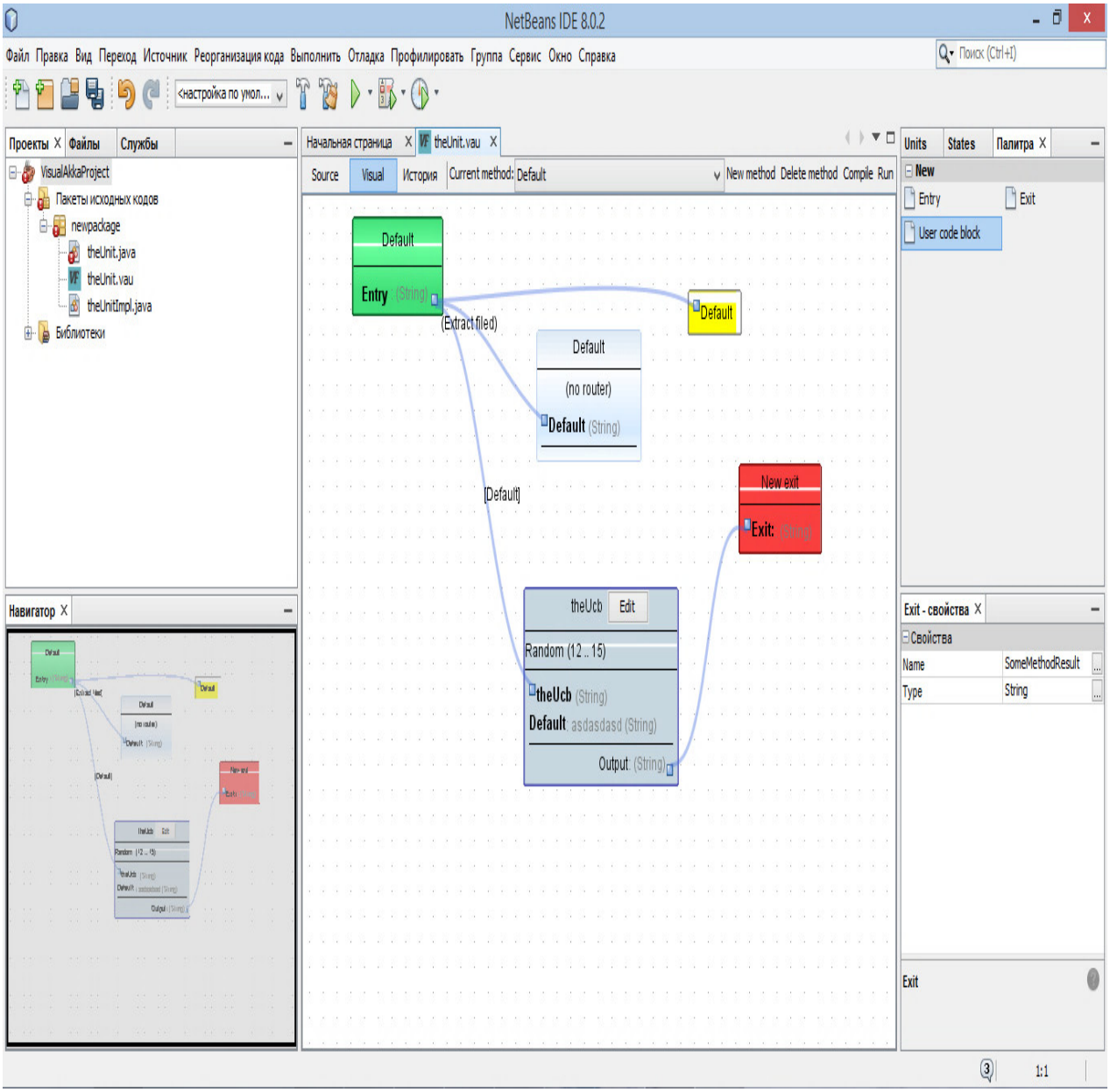

Рисунок 3.4 – Екранна форма плагіна на основі NetBeans IDE для Visual Akka

Редактор схем реалізовано на основі бібліотеки Visual Library. Кожна схема Visual Akka вдображає певний метод, який реалізує актора. Основними елементами схеми є входи, виходи, та екземпляри інших акторів. Кожен вхід є аргументом певного метода. Метод не буде викликаний, тобто повідомлення не розпочнуть рух по схемі доти, поки на кожному із входів не буде хоча б по одному повідомленні.

Вихід з'єднує поточного актора із іншими. Входи і виходи мають свій ідентифікатор та тип – будь який тип даних із мови Java, враховуючи обмеження фреймворку Akka.

Актори на схемі – це розгорнуті дочірні актори, поведінка яких визначається їх схемами. Схема актора може використовувати власний екземпляр, тобто реалізовувати рекурсивний виклик. Користувацькі модулі ззовні майже не відрізняються від моделей акторів, але їх суть зовсім різна. Користувацький модуль є реальним актором, який реалізує деякий конкретний метод відповідно до його специфікації на Java.

Приклад реалізації моделі акторів для перевірки даних на основі фреймворку Visual Akka продемонстровано на рисунку 3.5.

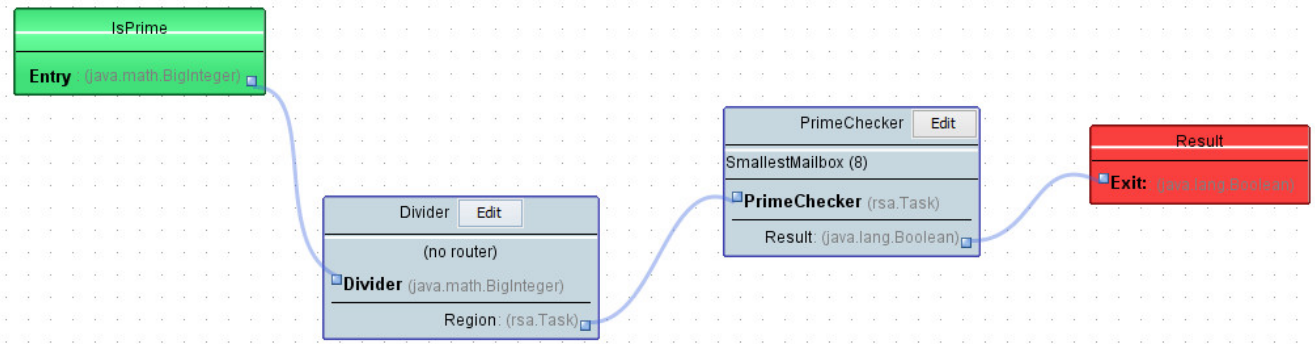

Рисунок 3.5 – Модель акторів для перевірки цифрових даних на простоту

З наведеного прикладу можна побачити, що кожен зв'язок на діаграмі відповідає одному виклику цільового актора. В параметрі виклику вказується аргумент чи метод, безпосередньо дані для передачі та типізований потік даних для повернення результату.

Для порівняння наведемо результати опрацювання даних на основі моделей акторів та існуючих класичних моделей, що реалізовують паралельні та розподілені обчислення. Ефективність методів реалізації паралельних обчислень обчислимо для різних наборів даних із максимальною кількістю векторів:

$$E = S / k ,$$

де S – показник прискорення;

k – кількість обробників.

Результати ефективності виконання алгоритмів реалізації паралельних обчислень у розподіленому середовищі наведено на рисунку 3.6.

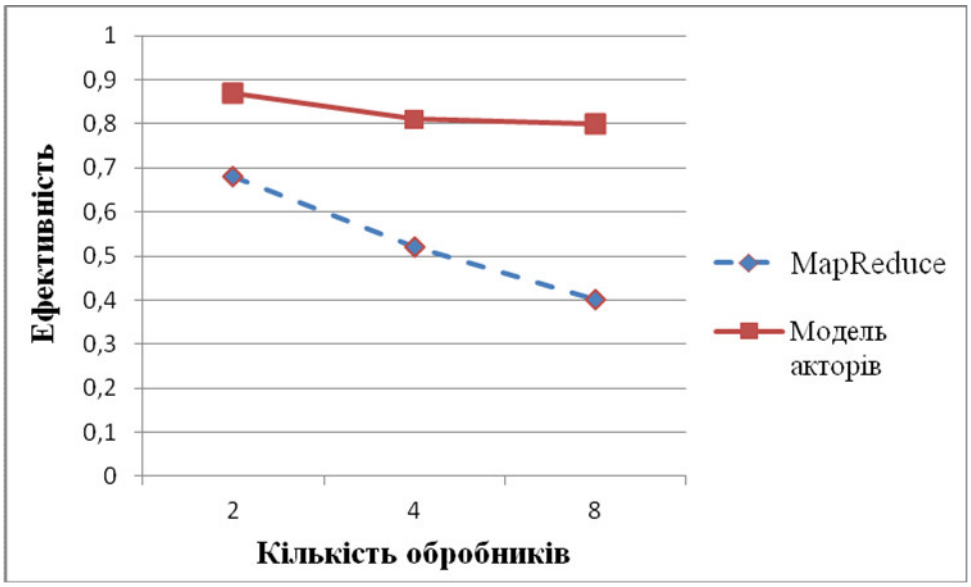

Рисунок 3.6 – Ефективність виконання алгоритмів у розподіленому середовищі

Як бачимо на рисунку 3.6, розроблена модель акторів ефективніша за існуючу модель MapReduce й при збільшенні кількості обробників даних (акторів-обробників) її продуктивність знижується не суттєво. Це свідчить про перспективність застосування акторних моделей при розробці розподілених систем.

4 ТЕХНІКО – ЕКОНОМІЧНЕ ОБҐРУНТУВАННЯ РОЗРОБКИ ПРОЕКТУ

4.1 Визначення витрат на оплату праці та відрахувань у соціальні фонди

Розроблене програмне забезпечення призначене для синтезу моделей акторів в межах предметно-орієнтованого підходу. Алгоритм синтезу моделей акторів для розробки розподілених систем, на якому базується розроблене програмне забезпечення, не вимагає значних обчислювальних процедур та характеризується високою швидкістю реалізації, що уможливлює одержати розподілену систему мінімальними затратами часу.

Витрати на розробку і впровадження програмних засобів (K) включають [16]:

$$K = K_1 + K_2, \tag{4.1}$$

де K_1 – витрати на розробку програмних засобів, грн.;

K_2 – витрати на відлагодження і дослідну експлуатацію програми вирішення задачі на комп’ютері, грн.

Витрати на розробку програмних засобів включають:

- витрати на оплату праці розробників;
- витрати на відрахування у спеціальні державні фонди;
- витрати на покупні вироби;
- витрати на придбання спецобладнання для проведення експериментальних робіт;
- накладні витрати;
- інші витрати.

Витрати на оплату праці включають заробітну плату (ЗП) всіх категорій працівників, безпосередньо зайнятих на всіх етапах проектування програмного засобу. Перелік необхідної програмної документації визначено відповідно до ДСТУ 3008-95 та включає:

– текст програми;

– керівництво користувача, яке включає інструкцію користувача;

– опис програми – відомості про логічну і фізичну модель, відомості щодо функціонування програми;

– пояснювальна записка – схема алгоритму, загальний опис алгоритму або функціонування програми, а також обґрунтовування прийнятих технічних і технічно-економічних рішень.

Перш за все визначаємо стадії розробки програмного засобу. У таблиці 4.1 відображено інформацію щодо етапів технологічного процесу розробки проекту

Таблиця 4.1 – Стадії розробки програмного засобу

№ п/п	Назва операції (стадії)	Виконавець, посада	Середній час виконання операції, год.
1	Підготовка, складання ТЗ	Менеджер проекту	3
2	Розробка макету системи	Team lead, Дизайнер (2)	2
3	Створення алгоритму системи	Архітектор (1)	3
4	Розробка системи	Програміст (6), дизайнер (2)	140
5	Тестування продукту	Тестувальник (3)	10
6	Попереднє представлення для замовника	Менеджер проекту	4
7	Представлення реалізованої системи	Менеджер проекту	1
Всього		14	163

Витрати на оплату праці розробників проекту визначаються за формулою:

$$B_{ОП} = \sum_{i=1}^{N} \sum_{j=1}^{M} n_{ij} \cdot t_{ij} \cdot C_{ij} \,, \qquad (4.2)$$

де n_{ij} – чисельність розробників *i*-ої спеціальності *j*-го тарифного розряду, осіб;

t_{ij} – затрачений час на розробку проекту співробітником *i*-ої спеціальності *j*-го тарифного розряду, год.;

C_{ij} – годинна ставка працівника *i*-ої спеціальності *j*-го тарифного розряду, грн.

За умов, якщо середньогодинну ставку розробника не відомо, її можна розрахувати за формулою:

$$C_{ij} = \frac{C_{ij}^{0}(1+h)}{РЧ_{i}}, \tag{4.3}$$

де C_{ij}^{0} – основна місячна заробітна плата розробника *i*-ої спеціальності *j*-го тарифного розряду, грн.;

h – коефіцієнт, що визначає розмір додаткової заробітної плати;

$РЧ_{i}$ – місячний фонд робочого часу працівника *i*-ої спеціальності *j*-го розряду, год.

Результати розрахунків записуємо у таблицю 4.2.

Таблиця 4.2 – Розрахунок витрат на оплату праці при розробці проекту

№ п/п	Посада виконавця	Час розробки, год.	Погодинна заробітна плата, грн	Витрати на оплату праці, грн
1	Back-end розробник (2)	25	350	17 500
2	Front-end розробник (2)	25	300	15 000
4	Team lead(1)	15	400	6 000
5	Python розробник(1)	35	350	12 250
6	Менеджер(1)	10	450	4 500
7	Тестувальник(3)	10	250	2 500
8	Дизайнер(2)	20	300	12 000
9	Архітектор(1)	3	225	675
Всього		143		70425

Оскільки виконавцем кваліфікаційної роботи є студент, то він є і розробником і тестувальником і дизайнером. Таким чином, оплата його праці – це стипендія без додаткових нарахувань, надбавок та премій. Стипендія студента становить 1400 грн. Зважачи на це, вартість проекту включає стипендію студента, а також витрати керівника на керівництво розробкою проекту та консультанта із написання техніко-економічного розділу. У таблицю 4.3 записуємо витрати на розробку програмного засобу у вигляді написання кваліфікаційної роботи.

Таблиця 4.3 – Розрахунок витрат на оплату праці

№ п/п	Посада виконавців	Час розробки, год	Погодинна заробітна плата, грн/год (за весь рік)	Витрати на розробку, грн.
1	Керівник КР, старший викладач	8	409	3272
2	Консультант з техніко-економічного розділу, старший викладач	1	409	409
3	Студент	150	22	3300
Разом				6981

У таблиці 4.4 наведено витрати на матеріали та комплектуючі вироби при виконанні кваліфікаційної роботи.

Загальна сума витрат на матеріальні ресурси B_M визначається за формулою:

$$B_M = \sum_{i=1}^{n} K_i \cdot Ц_i, \qquad (4.4)$$

де K_i - витрата *i*-го типу матеріалу, натуральні одиниці вимірювання;

$Ц_i$ - ціна за одиницю *i*-го типу матеріалу, грн.;

i - тип матеріального ресурсу;

n - кількість типів матеріальних ресурсів.

Таблиця 4.4 – Розрахунок витрат на матеріали та комплектуючі

№ п/п	Найменування купованих виробів	Одиниця виміру	Ціна, грн	Кількість куповани х виробів	Сума ,грн	Транспортні витрати (10% від суми)	Загальна сума, грн
1	Папір (формат А4)	уп	100,0	2	200,00	20,0	220,0
2	Ручка кулькова	шт	10,0	1	10,00	1,0	11,0
3	Олівець простий	шт	8	2	16,00	1,6	17,6
4	Диски CD-R	шт	9	2	18,00	1,8	19,8
5	Зошит, 24 арк.	шт	10	1	10	1	11
6	Тонер для принтера	уп	80	1	80	8,0	88
Разом							367,40

Накладні витрати включають три групи витрат: витрати на управління, загальногосподарські витрати, невиробничі витрати. Вони розраховуються за встановленими відсотками від витрат на оплату праці. При цьому накладні витрати складають 70% від заробітної плати:

$$H = 70425 \cdot 0,75 = 49297,5 \text{ грн.}$$

Інші витрати є витратами, які не враховані в попередніх статтях. Вони становлять 10% від заробітної плати:

$$I = 6981 \cdot 0,1 = 698,1 \text{ грн.}$$

Витрати на розробку програмного забезпечення складають:

$$K_1 = B_{ОП} + B_{Ф} + B_{КВ} + I . \quad (4.4)$$

$$K_1 = 6981 + 754,6 + 367,40 + 698,1 = 8801,1 \text{ грн.}$$

Витрати на відлагодження і дослідну експлуатацію програмного продукту визначаємо за формулою:

$$K_2 = S_{м.г.} \cdot t_{від}, \quad (4.5)$$

де $S_{м.г.}$ – вартість однієї машино-години роботи ПК, грн./год (приймаємо 6);

$t_{від}$ – час, витрачений на відлагодження і дослідну експлуатацію створеного програмного продукту, год.

Загальна кількість днів роботи на комп'ютері дорівнює 25 днів, середній щоденний час роботи на комп'ютері – 1 год., вартість години роботи комп'ютера дорівнює 6 грн. Звідси витрати на відлагодження та експлуатацію розраховуємо:

$$K_2 = 6 \cdot 25 = 150 \text{ грн.}$$

Оскільки розробка проекту включає застосування засобів обчислювальної техніки, то розрахуємо витрати на електроенергію, а результати розрахунків занесемо у таблицю 4.5.

Загальну суму витрат на електроенергію розраховуємо за формулою:

$$B_E = \sum_{i=1}^{\mathbf{n}} P_i \cdot k_i \cdot T_i \cdot Ц, \quad (4.6)$$

де P_i – паспортна потужність *i*-го електрообладнання, *кВт*;

k_i – коефіцієнт використання потужності *i*-го електрообладнання (приймається 0.7…0.9);

T_i – час роботи *i*-го устаткування за весь період розробки, год;

Ц – ціна електроенергії, грн / кВт*год;

i – тип електрообладнання;

n – кількість електрообладнання.

Таблиця 4.5 – Витрати на електроенергію

Назва устаткування	Паспортна потужність, кВт	Коефіцієнт використання потужності	Час роботи обладнання, год	Ціна Електроенергії грн, кВт/год	Сума, грн
Комп'ютер	0.5	0.7	163	1,68	95,85

До амортизації основних фондів включається сума амортизаційних відрахувань від вартості обладнання і приладів, що використовуються при розробці програмного продукту. Амортизаційні відрахування розраховуємо за формулою:

$$B_{AM} = \sum_{i=1}^{n} \frac{B_i \cdot H_i \cdot T_i}{100 \cdot T_{EФi}}, \qquad (4.7)$$

де B_i – вартість i-го устаткування, *грн.;*

H_i – річна норма амортизації i-го устаткування, %;

T_i – час роботи i-го устаткування за весь період розробки, год.;

$T_{EФi}$ – ефективний фонд часу роботи i-го устаткування за рік, год / рік;

i – тип устаткування;

n – кількість устаткування.

Таблиця 4.6 – Амортизація основних фондів

Найменування устаткування	Вартість устатку-вання, грн	Річна норма амортизації,%	Ефективний фонд часу роботи обладнання, год / рік	Час роботи обладнання для розробки системи, год	Сума, грн.
Комп'ютер	20000	60	1700	163	1150,6
Разом амортизаційні відрахування					1150,6

На основі отриманих даних в результаті обчислень складаємо кошторис витрат на розробку програмного забезпечення і заносимо їх у таблицю 4.7.

Таблиця 4.7 – Кошторис витрат на розробку програмного забезпечення

№ п/п	Найменування витрат	Сума витрат, грн.
1	Витрати на оплату праці	6981
2	Витрати на електроенергію	95,85
3	Витрати на куповані вироби	367,40
4	Амортизаційні відрахування	1150,6
5	Інші витрати	698,1
6	Витрати на відлагодження і дослідну експлуатацію програмного продукту	150
Разом		9442,95

4.2 Розрахунок ціни проекту

Для оцінки економічної ефективності розробленого програмного продукту слід порівняти його з аналогом, тобто існуючим програмним забезпеченням ідентичного функціонального призначення. Для цього визначимо експлуатаційні витрати на робробку проекту.

Експлуатаційні одноразові витрати по розробці програмного забезпечення і його аналогу включають вартість підготовки даних і вартість роботи комп'ютера (за час дії програми):

$$E_П = E_{1П} + E_{2П}, \tag{4.8}$$

де $E_П$ – одноразові експлуатаційні витрати на ПЗ (аналог), грн.;

$E_{1П}$ –вартість підготовки даних для експлуатації ПЗ (аналогу), грн.;

$E_{2П}$ – вартість роботи комп'ютера для розробки програмного продукту (аналогу), грн.

Річні експлуатаційні витрати $B_{ЕП}$ визначаються за формулою:

$$B_{ЕП} = E_{П} * N_{П}, \quad (4.9)$$

де $N_{П}$ – періодичність експлуатації ПЗ (аналогу), раз/рік.

Вартість підготовки даних для роботи на комп'ютері визначається за формулою:

$$E_{1П} = \sum_{l=1}^{n} n_i t_i c_i, \quad (4.8)$$

де i – категорії працівників, які приймають участь у підготовці даних (i=1,2,...n);

n_i – кількість працівників i-ої категорії, осіб;

t_i – трудомісткість роботи співробітників i-ої категорії по підготовці даних, год.;

c_i – середньогогодинна ставка працівника i-ої категорії з врахуванням додаткової заробітної плати, що знаходиться із співвідношення (4.3).

Трудомісткість підготовки даних для проектного рішення складає 4 год., для аналога 2 год.

Таблиця 4.7 – Розрахунок витрат на реалізацію програмного забезпечення

№	Час роботи співробітників, год.	Середньогодинна заробітна плата, грн./год.	Витрати , грн.
	Проектне рішення		
1	4	8	32
	Аналог		
1	2	32	64

Витрати на експлуатацію комп'ютера визначається за формулою:

$$E_{2П} = t * S_{МГ},$$

де t – витрати машинного часу для реалізації проектного рішення (аналогу), год.;

$S_{МГ}$ – вартість однієї години роботи комп'ютера, грн./год.

Зважаючи на вищенаписане, проверемо розрахунки:

$$E_{2П} = 4 \cdot 6 = 24 \text{грн.},\ E_{2П_a} = 2 \cdot 6 = 12 \text{грн.};$$

$$E_{П} = 32 + 24 = 56 \text{ грн.},\ E_{П_a} = 64 + 12 = 78 \text{ грн.};$$

$$B_{ЕП} = 56 \cdot 252 = 14112 \text{ грн.},\ B_{ЕП_a} = 78 \cdot 252 = 19656 \text{грн.}$$

Ціна програмного продукту – це витрати на придбання і експлуатацію програмного засобу за весь період його служби:

$$Ц_{П} = К \cdot (1 + \frac{П_{Р}}{100}) + К_0 + К_к, \qquad (4.9)$$

де $К$ – кошторисна вартість;

$П_р$ – рентабельність;

$К_0$ – витрати на встановлення та освоєння програмного засобу на конкретному об'єкті, грн.;

$К_к$ – витрати на доукомплектування технічних засобів на об'єкті, грн. Зважаючи на вищеописане, розрахуємо ціну програмного засобу

$$Ц_{П_a} = 141719{,}525 \cdot (1 + 0{,}3) = 184235{,}4 \text{ грн.}$$

$$Ц_{П} = 9442{,}95 \cdot (1 + 0{,}3) = 12275{,}8 \text{ грн.}$$

У наступному підрозділі проведемо аналіз економічної ефективності

розробки програмного продукту.

4.3 Визначення економічної ефективності розробки проекту

Для того, щоб побудувати таблицю показників економічної ефективності розробки програмного продукту, проведемо розрахунки необхідних показників.

Економічний ефект в сфері проектування проектного рішення розраховуємо за формулою:

$$E_{ПР} = Ц_{П} - Ц_{А}, \tag{4.10}$$

$$E_{ПР} = 184235{,}4 - 12275{,}8 = 171959{,}6 \text{ грн.}$$

Річний економічний ефект від експлуатації програмного продукту:

$$E_{КС} = B_{ЕА} - B_{ЕП},\ E_{КС} = 19656 - 14112 = 5544 \text{ грн.} \tag{4.11}$$

Дохід від розробки ПЗ у *i*-му періоді розраховуємо за формулою:

$$D_i = J_i(B_i - C_i), \tag{4.12}$$

де B_i – ціна продажу програмного продукту в *i*-му періоді;

C_i – собівартість програмного продукту (фактично дорівнює сумі витрат на розробку ПЗ);

J_i – кількість ПЗ.

$$D_i = 1 \cdot (12275{,}8 - 9442{,}95) = 2832{,}85 \text{ грн.}$$

Економічний ефект полягає у відношенні результату від розробленого програмного продукту до затрачених ресурсів та розраховується за формулою:

$$E = \frac{D_i}{B_{заг}}. \quad (4.13)$$

$$E = \frac{2832{,}85}{9442{,}95} = 0{,}3.$$

Тоді термін окупності обчислюємо за такою формулою:

$$T = \frac{1}{E} \quad (4.14)$$

$$T = \frac{1}{0{,}3} = 3{,}3 \text{ р.}$$

Зважаючи на проведені розрахунки ефективності розробки програмного продукту, обчислимо сумарний ефект від розробки програмного продукту за формулою:

$$E = E_{ПР} + E_{КС}$$

$$E = 171959{,}6 + 5544 = 177503{,}6 \text{ грн.}$$

Таблиця 4.8 – Показники економічної ефективності проектного рішення

№	Найменування	Значення показників	
		Аналог	Новий варіант
1	Капітальні вкладення	70425	9442,95
2	Ціна придбання	184235,4	12275,8
3	Економічний ефект в сфері проектування	-	171959,6
4	Економічний ефект в сфері експлуатації	-	5544
5	Дохід від розробки		2832,85
6	Сумарний ефект	177503,6	
7	Термін окупності проекту	3,3	

Отже, у цьому розділі проведено розрахунок витрат на розробку програмного забезпечення. Показники, що характеризують витрати на розробку програмного продукту порівняно із показниками, які характеризують програмний продукт із аналогічним функціональним призначенням.

Розроблене програмне забезпечення має суттєві переваги у порівнянні із аналогами, зокрема простота використання, швидкість проведення розрахунків та достатній ступінь паралелізму.

Згідно із проведеними розрахунками, що обґрунтовують економічну ефективність розробки програмного продукту, можна зробити висновок, що розроблене програмне забезпечення є суттєво дешевшим, оскільки у ролі розробника виступає студент. Отримано економічний ефект у розмірі 177503,6 грн., що свідчить про економічну доцільність розробки і впровадження програмного забезпечення для синтезу моделей акторів на основі предметно-орієнтованого підходу при розробці розподілених систем.

ВИСНОВКИ

1. Охарактезизовано особливості та основні відмінності розробки програмних продуктів на основі предметно-орієнтованого підходу. Зазначено, що мови програмування в межах підходу орієнтуються на предметну область задачі.

2. Проаналізовано існуючі програмні засоби в межах предметно-орієнтованого підходу й зазначено, що в більшості реалізація моделей на їх основі вимагає складних обчислювальних процедур та із збільшенням кількості даних суттєво програють у продуктивності.

3. Описано методи розробки розподілених систем й виокреслено їх основні недоліки, зокрема, труднощі із типізацією даних, асинхронізація потоків виконання операцій.

4. Проаналізовано моделі на основі базових структурних елементів – акторів. Такі моделі здатні опрацьовувати великі об’єми даних на основі посилання повідомлень між акторами, прості у реалізації, не вимагають значних обчислювальних ресурсів.

5. Зазначено, що існуючі програмні засоби реалізації акторних моделей обмежено враховують властивосі моделей акторів, оскільки не враховують даних із спільною пам’яттю. Відзначено, що фреймворк Akka містить необхідні механізми для реалізації акторних моделей.

6. Розроблено алгоритм синтезу моделей акторів на основі візуального підходу, що простий у реалізації, відзначається швидкодією, оскільки Visual Akka містить додатковий рівень абстракції та забезпечує статичну типізацію повідомлень.

7. Розроблено акторну модель для організації даних у розподіленому середовищі. Результатами експериментів підтверджено перспективність її застосування для розв’язуваня подібних задач.

8. Обгрунтовано техніко-економічні показники доцільності розробки проекту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бабичев С. Л., Коньов К. А. Распределенные системы : учебное пособие для вузов. М.: Издательство Юрай, 2019. 507 с.
2. Баканов В. М. Параллельные вычисления: Учебное пособие. М.: МГУПИ, 2006. 124 с.
3. Барабаш О. В., Куліковська Ю. А. Аналіз експлуатаційних проблем розподілених систем // Наукові записки Українського науково-дослідного інституту зв'язку. 2015 р. Вип.4 (38). С. 86-94.
4. Бычков И. В., Опарин Г. А., Новопашин А. П., Сидоров И. А., Горский С. А. Методы и средства организации параллельных и разпределенных вычислений на основе парадигмы модульного программирования // Вестник Кемеровского государственного университета. 2012.Т. 2. №4 (52). С. 22-30.
5. Борис Т. В., Алєксєєв М. О. Порівняльний аналіз технології паралельного обчислення великих масивів даних MapReduce // Proceedings of Second International Conference "Cluster Computing". Lviv , 2013. С. 54-57.
6. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. 608 с.
7. Гафаров Ф.М., Галимянов А.Ф. Параллельные вычисления: учеб. пособие. Казань: изд-во Казан. ун-та, 2018. 149 с.
8. Галкін О. В., Верес М. М., Ларін В. О., Бантиш О. В. Використання предметно–орієнтованої мови і візуальних підходів для проектування системи акторів (AKKA) // Проблеми інформаційних технологій. 2018. № 1. С. 148-153.
9. Гергель В. П. Теория и практика параллельных вычислений. М.: БИНОМ, 2007. 424 с.
10. Гладкий М. В. Модель распределенных вычислений MapReduce // Труды БГТУ. 2016. Т. № 6. С. 194-198.

11. Глибовець М. М., Зінчук С. О. Використання моделі акторів для реалізації розподілених обчислень // Системні дослідження та інформаційні технології. 2015. № 2. С. 16-25.

12. Глибовець А.Н. Решение задачи распределенного индексирования в оперативной памяти на базе модели актеров с использованием фреймворка Akka // УСиМ. 2014. № 4. С. 61-72.

13. Демина А. В. УСиМ, 2014, № 4 Н. Распределенные системы : учебное пособие для студентов. Саратов: ССЕИ им.Г.В. Плеханова, 2018. 108 с.

14. Ежова Н. А., Соколинский Л. Б. Модель параллельных вычислений для многопроцессорных систем с распределенной памятью // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2018. № 2. Т. 7. С. 32-49.

15. Забродин А. В., Луцкий А. Е. Параллельные вычисления при решении современных задач науки и техники: веб-сайт. URL: http://compmech.math.msu.su/publ/Zabrodin_Parallel_comp.pdf (дата звернення 13 08 2020).

16. Казакова А.С. Методы и инструменты реализации предметно-ориентированных языков программирования: веб-сайт. URL: http://sysprog.info/2009/03.pdf (дата звернення 12.04.2021).

17. Кознов Д.В. Основы визуального моделирования. Интернет-университет информационных технологий; БИНОМ. Лаборатория Знаний, 2008. 246 с.

18. Косяков М. С. Введение в распределенные вычисления. Санкт-Петербург: НИУ ИТМО, 2014. 155 с.

19. Литвиненко В. А., Хованских С. А. Решения задач путем организации разпределенных вычислений в сети // Известия Южного федерального университета. Технические науки. 2008. С. 16-21.

20. Лорин Г. Распределенные вычислительные системы. М.: Радио и связь, 1984. 296 с.

21. Митяков А. В., Татаринов Ю. С. Подходы к эффективному исполнению итеративных алгоритмов на модели MapReduce // Известия СПбГЭТУ «ЛЭТИ». 2014. Т. № 4. С. 8-14.

22. Модель Акторов и C++: что, зачем и как?: веб-сайт. URL: https://habr.com/ru/post/322250/ (дата звернення 15. 10. 2020).

23. Методичні вказівки до виконання практичних робіт з дисципліни «Техніко-економічне обґрунтування розробки комп'ютерних систем"/ Н.Я. Савка, І.Р. Паздрій / Під ред. О.М. Березького. Тернопіль: ТНЕУ, 2019. 40 с.

24. Методичні вказівки до оформлення курсових, звітів про проходження практики, випускних кваліфікаційних робіт для студентів спеціальності «Комп'ютерна інженерія» / І.В. Гураль, Л.О.Дубчак / під ред. О.М. Березького. Тернопіль: ТНЕУ, 2019. 34 с.

25. Методичні рекомендації до виконання кваліфікаційної роботи з освітнього ступеня "Бакалавр" спеціальності 123 «Комп'ютерна інженерія» галузі знань 12 Інформаційні технології / О.М. Березький, Л.О.Дубчак, Г.М. Мельник, Ю.М. Батько / Під ред. О.М. Березького. Тернопіль: ЗУНУ, 2020. 60с.

26. Петухов И. В. Использование модели акторов для параллельного выполнения алгоритмов интеллектуального анализа данных, основанных на блоковой структуре // Известия СПбГЭТУ «ЛЭТИ». 2015. С. 38-42.

27. Пєчкурова О. М., Ахмедзянов П. А. Аналіз фреймворків реалізації моделі акторів // Наукові записки. 2015 р. Т. 177. С. 125-130.

28. Савка Н.Я., Складанюк В.М.

29. Старченко А.В. Берцун В.Н. Методы параллельных вичислений. Томск: Изд-во Том. ун-та, 2013. 223 с.

30. Стрига Д. М., Кудінова А. О. Паралельне програмування на мові ERLANG: веб-сайт. URL: https://www.ukrlogos.in.ua/10.11232-2663-4139.04.34.html (дата звернення 12. 04. 2021).

31. Таненбаум Э. Распределенные системы: принципы и парадигмы. Питер: СПб, 2003. 877 с.

32. Тель Ж. Введение в распределенные алгоритмы. М.: МЦНМО, 2009. 616 с.

33. Топорков В. В. Модели распределенных вычислений. М.: Физматлит, 2004. 320 с.

34. Хьюз К., Хьюз Т. Параллельное и распределенное программирование C++. М. : Издательский дом «Вильямс», 2004. 672 с.

35. Якевлєв Д.А. Модели актера для реализации надежных, отказоустойчивых систем: веб-сайт. URL: https://cyberleninka.ru/article/n/modeli-aktera-dlya-realizatsii-nadezhnyh-otkazoustoychivyh-sistem/viewer (дата звернення 19.04.2021).

36. Ясько М. М. Паралельна обробка даних. Дніпропетровськ: РВВ ДНУ, 2010. 76 с.

37. Akka, акторы и реактивное программирование): веб-сайт. URL: https://habr.com/ru/company/piter/blog/266103/ (дата звернення 18.05.2021).

38. Akka: веб-сайт. URL: https://akka.io/ (дата зверненя 19.05. 2021).

39. Amaris M. A. Simple BSP-based Model to Predict Execution Time in GPU Applications // Proceedings of 22nd International Conference on High Performance Computing (HiPC). IEEE, 2015. P. 285–294.

40. ANTLR: неформальное введение): веб-сайт. URL: https://dou.ua/lenta/articles/antlr-introduction/ (дата звернення 27.11.2020).

41. Codemodel: веб-сайт. URL: https://javaee.github.io/jaxb-codemodel/ (дата звернення 28 07 2020).

42. Cordy J.R. The TXL Source Transformation Language // Science of Computer Programming. Vol.61. Issue 3. 2006. P.190–210.

43. E: веб-сайт. URL: http://progopedia.ru/language/e/ (дата звернення 17 06 2020).

44. Fowler M. Domain-Specific Languages. Addison-Wesley Professional. USA, 2010. 582 с.

45. MapReduce – краткое руководство: веб-сайт. URL: https://coderlessons.com/tutorials/bolshie-dannye-i-analitika/izuchit-kartu-umenshit/mapreduce-kratkoe-rukovodstvo (дата звернення 15.03.2021).

46. Scholliers C., Tanter E., De Meuter W. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model // Science of Computer Programming. 2014. Vol. 80. P. 133–138.

47. TOUR OF SCALA: веб-сайт. URL: https://docs.scala-lang.org/ru/tour/tour-of-scala.html (дата звернення 18.05.2021).

48. Van Den Brand M.G.J., Heering J., Klint P., Oliver P. Compiling Language Definitions: the ASF+SDF Compiler // ACM Transactions on Programming Languages and Systems. Vol.24, N 4. 2002. P.334– 368.

49. Vernon V. Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Addison-Wesley Professiona, 2015. 486 p.

50. Welsh N. Why I Don't Like Akka Actors: веб-сайт. URL: https://noelwelsh.com/posts/2013-03-04-why-i-dont-like-akka-actors.html (дата звернення 19 06 2020).