МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХЕРСОНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП’ЮТЕРНИХ НАУК, ФІЗИКИ ТА МАТЕМАТИКИ

РОЗРОБКА АЛГОРИТМУ ТРАНСЛЯЦІЇ ВІЗУАЛІЗАЦІЇ АЛГЕБРАЇЧНОГО ПРЕДСТАВЛЕННЯ ПОВЕДІНКИ ІНСЕРЦІЙНОГО МОДЕЛЮВАННЯ

Кваліфікаційна робота (проєкт)

на здобуття ступеня вищої освіти «магістр»

Виконав: студент 2 курсу 261М групи
Спеціальності 122 Комп'ютерні науки
Кушніренко Олександр Борисович
Керівник: д-р. фіз-мат наук, професор
Песчаненко В.С.

Рецензент: Тарасіч Ю.Г., докторант інституту кібернетики імені В.М. Глушкова НАН України, керівник компанії «Garuda.AI»

Івано-Франківськ – 2023

ЗМІСТ

ВСТУП..3
РОЗДІЛ 1. Інсерційне моделювання..5
1.1. Системи переписування термів...5
1.2. Система алгебраїчного програмування..................................10
1.3. Система інсерційного моделювання.......................................12
РОЗДІЛ 2. Сучасні технології створення інтерфейсів користувача в інтернет...23
2.1. JavaScript та web-framework React.js.......................................23
2.2. Компоненти React, та їх взаємодія..30
2.3. Життєвий цикл React застосунків...33
2.4. Маршрутизація у React — застосунках..................................34
2.5. Тестування компонентів React...35
РОЗДІЛ 3. Система візуалізації графічного представлення поведінки моделі у системі інсерційного моделювання.............39
3.1. React застосунок для відображення поведінки моделі інсерційного моделювання..39
ВИСНОВКИ
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ARS — абстрактна система редукцій

АПС (APS) — система алгебраїчного програмування

IMS — система інсерційного моделювання

delta — спеціальне позначення (ключове слово), яке використовується системою інсерційного моделювання для позначення стану успішного завершення роботи.

Deadlock — спеціальне позначення (ключове слово), яке використовується системою інсерційного моделювання для позначення "тупікового стану"

SDL — мова специфікацій із формальною семантикою

MSC — стандартизована діаграма взаємодії для мови SDL

JSON – формат обміну даними в інтернеті,який представляє собою текстове представлення об'єктів мови javascript

ВСТУП

Система інсерційного моделювання активно розвивається протягом останніх кількох десятків років. Заснована на попередній системі алгебраїчного програмування, вона дозволяє досліджувати складні розподілені системи, та застосовувати до них методи формальної верифікації програмного забезпечення. Подібний підхід (представлення системи у вигляді взаємодіючих агентів та середовищ) у наш час використовують багато провідних компаній, що займаються розробкою програмного забезпечення (наприклад обчислення мобільних амбієнт Л.Карделлі, що використовуються компанією Microsoft для верифікації взаємодіючих систем). Для оптимізації розробки моделей для їх подальшого використання, та для наочності системі потрібний зручний та привабливий інтерфейс.

Актуальність — існуючий інтерфейс користувача системи інсерційного моделювання — це інтерфейс звичайного консольного застосунка. Такий вигляд результатів моделювання задовольняє потреби професіонального програміста з інсерційного моделювання, проте не підходить для презентації результатів роботи моделі, або для вивчення системи. Тому розробка наглядного інтерфейсу з використанням сучасних графічних фреймворків, які окрім того також можуть працювати через мережу інтернет є актуальною.

Метою даної роботи є створення сучасного та зручного інтерфейсу для відображення результатів роботи моделі у системі інсерційного моделювання, який може бути використаний як у локальній системі так і через мережу інтернет.

Згідно до поставленої мети можна виділити наступні **задачі**:

1. Створити інтернет застосунок, використовуючи технологію ReactJS, який у вигляді графа відображатиме дерево станів та переходів, отримане в результаті роботи інсерційної моделі.
2. Розробити транслятор, що перетворить файл з результатами роботи інсерційної машини у JSON – формат необхідний фреймворку React.
3. Розробити серверну частину застосунка, яка буде відповідати за зв' язок між інсерційною машиною та React застосунком, тобто перетворення формату алгебри поведінок у формат JSON, використовуючі технологію Node.JS.
4. Створити коллекцію модульних тестів (unit tests), для підтримки подальшого розвитку та розширення функціональності системи візуалізації результатів роботи інсерційної моделі.

Об'єкт дослідження — результати роботи моделі системи інсерційного моделювання.

Предмет дослідження — формат виводу результатів роботи системи інсерційного моделювання.

Наукова новізна одержаних результатів полягає в тому, що наглядне графічне відображення результатів роботи інсерційної моделі у значній мірі сприятиме кращому сприйняття результатів роботи моделі, та зменшить час необхідний для їх аналізу.

Практичне значення одержаних результатів полягає в тому, що графічне відображення результатів роботи інсерційної моделі дозволяє скоротити час, необхідний для обробки та аналізу результатів, зокрема пошуку *deadlock* станів та *delta* станів.

РОЗДІЛ 1. АЛГЕБРАЇЧНЕ ПРОГРАМУВАННЯ ТА ІНСЕРЦІЙНЕ МОДЕЛЮВАННЯ

1.1. Системи переписування термів

У математиці, інформатиці та логіці переписування охоплює широкий спектр методів заміни під-термів формули іншими термами. Такі методи можуть бути досягнуті за допомогою систем переписування (разом із терміном системи переписування часто використовують такі визначення як механізми переписування або системи редукцій). У своїй основній формі вони складаються з набору об’єктів, а також сукупності певних правил щодо того, як трансформувати ці об’єкти.

Переписування може бути недетермінованим. Одне правило для переписування терма може бути застосоване багатьма різними стратегіями до цього терма, також може бути застосовано кілька правил. Таким чином, системи переписування надають не алгоритм зміни одного терма на інший, а набір правил можливого перетворення цього тему в інший [2]. Однак у поєднанні з відповідним алгоритмом, який називається стратегією переписування, системи переписування можна розглядати як комп’ютерні програми. Деякі системи автоматичного доведення теорем і декларативні мови програмування засновані на переписуванні термів.

Абстрактна система переписування (також у літературі можна зустріти наступні визначення: система редукції або абстрактна система перезапису) — це формалізм, який охоплює основні поняття та властивості таких систем [3]. У своїй найпростішій формі абстрактна система переписування — це просто набір («об’єктів») разом із бінарним відношенням, яке традиційно позначають як →. Таке визначення може бути додатково уточнено, якщо підмножини бінарного

відношення буде проіндексовано. Незважаючи на свою простоту, абстрактної системи переписування повністю достатньо для опису важливих властивостей цих систем, таких як нормальні форми, закінчення та різні поняття злиття.

Історично склалося кілька формалізацій переписування в абстрактному середовищі, кожна зі своїми особливостями. Частково це пов'язано з тим, що деякі поняття є еквівалентними. Формалізація, яка найчастіше зустрічається в монографіях і підручниках і якої тут зазвичай дотримуються, належить Жерару Юе (1980)

Абстрактна система редукції (ARS) — це найзагальніше (одновимірне) поняття про специфікацію набору об'єктів і правил, які можна застосувати для їх перетворення [3]. Останнім часом автори також використовують термін абстрактна система переписування.(Надання переваги слову «зменшення» тут замість «переписування» означає відхід від однакового використання «переписування» в назвах систем, які є конкретизаціями абстрактна система редукції [14]. Оскільки слово «зменшення» не зустрічається в назвах більш спеціалізовані системи, у старих текстах система скорочення є синонімом до виразу абстрактна система редукції.

Абстрактна система редукції — це множина, елементи якого зазвичай називаються об'єктами разом із бінарним відношенням на A, яке традиційно позначається → і називається відношенням скорочення, відношенням переписування або просто скороченням. Ця (вкорінена) термінологія з використанням «зменшення» трохи вводить в оману, оскільки відношення не обов'язково зменшує певну міру об'єктів.

У деяких контекстах може бути корисним розрізняти деякі підмножини правил, тобто деякі підмножини відношення редукції →,

напр. все відношення редукції може складатися з правил асоціативності та комутативності [4]. Отже, деякі автори визначають редукційне відношення → як індексоване об'єднання деяких відношень.

Як математичний об’єкт абстрактна система редукції є точно таким же, як і система переходів між станами без міток, і якщо відношення розглядається як індексоване об’єднання, тоді абстрактна система редукції є такою самою, як і система переходів із мітками, де індекси є мітками. Однак фокус дослідження та термінологія відрізняються. У системах переходів між станами найбільша увага приділяється до інтерпретації міток як дій, тоді як у абстрактній системі редукцій фокус зосереджений на тому, як об’єкти можуть бути перетворені (переписані) в інші.

В системах правил переписування дуже важливу роль відіграє поняття нормальної форми. Кажуть що об’єкт знаходиться в нормальній формі, якщо до нього більше неможливо застосувати правила переписування, тобто він є незмінним. Залежно від системи переписування, об’єкт можна привести одразу до кількох нормальних форм або взагалі неможливо до жодної. Багато важливих властивостей систем переписування стосуються нормальних форм [7].

Формально, якщо $(A,\rightarrow)$ є абстрактною системою переписування, то $x \in A$ є нормальною формою, якщо не існує $y \in A$ такого, що $x \rightarrow y$, тобто x є незмінним доданком.

Об’єкт a є слабо нормалізованим, якщо існує принаймні одна конкретна послідовність переписувань, починаючи з a, яка в кінцевому підсумку дає нормальну форму. Система перезапису має властивість слабкої нормалізації або є (слабко) нормалізованою (WN), якщо кожен об’єкт слабо нормалізується. Об’єкт a є сильно нормалізованим, якщо

кожна послідовність переписувань, починаючи з *a*, врешті-решт завершується нормальною формою. Абстрактна система переписування є сильно нормалізовною, завершеною, недетермінованою або має властивість (сильної) нормалізації (SN), якщо кожен з її об'єктів є сильно нормалізованим.

Система переписування має властивість нормальної форми (NF), якщо для всіх об'єктів *a* і нормальних форм *b*, *b* може бути виведеним з а серією переписувань та зворотних перетворень, лише якщо *a* зводиться до *b*. Система перезапису має унікальну властивість нормальної форми (UN), якщо для всіх нормальних форм а, $b \in S$, а може бути досягнуто з *b* серією переписувань та зворотних перетворень, лише якщо а дорівнює *b*. Система переписування має унікальну властивість нормальної форми щодо редукції (UN→), якщо для кожного терма, що зводиться до нормальних форм *a* і *b*, а дорівнює b. Також слід окремо визначити поняття канонічної форми [11].

У математиці та інформатиці канонічна, нормальна або стандартна форма математичного об'єкта є стандартним способом представлення цього об'єкта як математичного виразу. Часто це те, що забезпечує найпростіше представлення об'єкта та дозволяє ідентифікувати його унікальним способом. Різниця між «канонічною» та «нормальною» формами варіюється від підполя до підполя. У більшості полів канонічна форма визначає унікальне представлення для кожного об'єкта, тоді як нормальна форма просто визначає його форму без вимоги унікальності.

Наприклад канонічною формою натурального числа в десятковому представленні є кінцева послідовність цифр, яка не починається з нуля. Загалом, для класу об'єктів, на якому визначено

відношення еквівалентності, канонічна форма полягає у виборі конкретного об’єкта в кожному класі. Наприклад:

Жорданова нормальна форма — це канонічна форма подібності матриці.

Форма рядкового ешелону — це канонічна форма, коли еквівалентними вважаються матриця та її лівий добуток на оборотну матрицю [3].

В інформатиці, а точніше в комп’ютерній алгебрі, при представленні математичних об’єктів у комп’ютері зазвичай існує багато різних способів представлення того самого об’єкта. У цьому контексті канонічна форма — це таке подання, що кожен об’єкт має унікальне подання (з канонізацією — це процес, за допомогою якого подання вводиться в його канонічну форму). Таким чином, рівність двох об'єктів можна легко перевірити шляхом перевірки рівності їх канонічних форм.

Незважаючи на цю перевагу, канонічні форми часто залежать від довільного вибору (наприклад, упорядкування змінних), що створює труднощі для перевірки рівності двох об’єктів, що випливає з незалежних обчислень. Тому в комп’ютерній алгебрі нормальна форма є слабшим поняттям: нормальна форма — це представлення, яке представляє нуль однозначно. Це дозволяє перевірити рівність, помістивши різницю двох об’єктів у нормальну форму [13].

Канонічна форма також може означати диференціальну форму, яка визначена природним (канонічним) способом.

У практичному плані часто вигідно мати можливість розпізнавати канонічні форми. Існує також практичне, алгоритмічне питання, яке має бути розглянуто: як перейти від даного об’єкта s в S до його канонічної

форми s^*? Канонічні форми зазвичай використовуються, щоб зробити роботу з класами еквівалентності більш ефективною. Наприклад, у модульній арифметиці канонічна форма для класу залишків зазвичай береться як найменше невід'ємне ціле число в ньому. Операції над класами виконуються шляхом комбінування цих представників, а потім зведення результату до його найменшого невід'ємного залишку. Вимога унікальності іноді пом'якшується, дозволяючи формам бути унікальними аж до більш тонкого відношення еквівалентності, наприклад, дозволяючи змінювати порядок термінів (якщо немає природного порядку термінів) [15].

Канонічна форма може бути просто конвенцією або глибокою теоремою. Наприклад, поліноми традиційно записуються з доданками в спадних степенях: більш звично писати , ніж , хоча ці дві форми визначають той самий поліном. Навпаки, існування жорданової канонічної форми для матриці є глибокою теоремою.

1.2. Система алгебраїчного програмування

Алгебраїчне програмування - це програмування, засноване на переписуванні. Алгебраїчне програмування є розширенням функціонального програмування та застосовується при вирішенні завдань комп'ютерної алгебри (таких як проблема слів у певних алгебрах, алгоритми поповнення Кнута-Бендикса або Бухбергера), а також задач, пов'язаних із операційною семантикою мов програмування (виконання алгебраїчних специфікації компонентів програмного забезпечення, визначення операційних семантик мов програмування, розробка інтерпретаторів та прототипів компонентів програмного забезпечення та ін.) [4].

На відміну від традиційного підходу, орієнтованого на використання канонічних систем правил переписування з «очевидними» стратегіями їх застосування, в АПС можливо поєднання будь-яких систем правил переписування та різноманітних стратегій переписування.

Такий підхід значно розширює можливості техніки переписування, оскільки зростає їх гнучкість та виразність. Система алгебраїчного програмування інтегрує чотири основні парадигми програмування таким чином, що основна частина програми може бути написана у вигляді системи правил переписування, імперативне та функціональне програмування використовуються для визначення стратегій, парадигма логічного програмування реалізується на основі переписування, використовуючи вбудовану процедуру уніфікації. [2]

У загальному вигляді (узагальнена форма Бекуса-Наура) система правил переписування виглядає наступним чином (1.1).

<система правил переписування>::=rs(<список змінних>)

(<список правил>)

<правило>::=<просте правило> | <умовне правило>

<просте правило>::=<алгебраїчний вираз>=<алгебраїчний вираз>

<умовне правило>::=<умова> → <просте правило>

<змінна>::=<ідентифікатор> (1.1)

Систему алгебраїчного програмування було розроблено кафедрою 100,105 Глушкова Інститут кібернетики НАН України у 1987 р. Вона стала першою системою переписування термінів, яка використовувала систему правил і стратегій переписування окремо.[4]

Система Алгебраїчного Програмування об'єднує чотири основні парадигми програмування наступним чином. Основна частина програми може бути написана у вигляді системи правил переписування. Використовується імперативне та функціональне програмування для визначення стратегій. Логічна парадигма реалізована на основі переписування за допомогою вбудованої процедури уніфікації [12].

Було три реалізації системи алгебраїчного програмування APS, а саме в 1987 році(APS v.1), 2004 (APS v.2), 2009 (APS v.3).

1.3. Система Інсерційного моделювання

Інсерційне моделювання — це технологія проектування системи, заснована на теорії взаємодії агентів та середовищ. Ця теорія заснована на алгебрі процесів і призначена для уніфікації різних моделей взаємодії та обчислень (таких як CCS, CSP, π-calculus, mobile ambients тощо). В останні роки цей підхід був успішно застосований до проблеми перевірки специфікацій вимог для розподілених паралельних систем з різні предметні області, включаючи телекомунікації, телематику, розподілені обчислення та ін [17].

Ці програми підтримуються системою VRS, розробленою для Motorola компанією Kiev VRS-група. У поєднанні з системою TAT, розробленою в Motorola Software Group Russia, вона також підтримує генерацію тестових випадків із специфікацій вимог.

Система використовує основні специфікації базових протоколів для формалізації специфікацій вимог до розподілених паралельних систем. Базові протоколи – це параметризовані MSC (діаграми послідовності повідомлень) з передумовами та постумовами, які

інтерпретуються відповідно до станів середовища із зануреними у нього агентами [19].

Математично базовий протокол можна розглядати як оператор певного типу динамічної логіки, що має назву трійки Хоара (1.2).

$$\forall x \ (a \rightarrow \langle P \rangle \rightarrow b) \qquad (1.2)$$

У цьому операторі *x* — список типізовании̂ параметрів, *a* і *b* — передумова і постумова відповідно, а *P* – процес, визначений діаграмою послідовності повідомлень. Передумови та постумови — це формули багатосортової мови першого порядку, яка називається базовою мовою. Ця мова використовується для опису властивостей станів системи, представленої у вигляді композиції середовищ та агенти, які занурені (вставлені) в це середовище. Частина системи, що змінюється та розвивається, представлена у вигляді функціональних та предикатних символів основної мови під назвою атрибути середовища. Процес *u* описує кінцеву поведінку середовища із зануреними (вставленими) агентами. Коли параметри базового протоколу фіксовані, то можна говорити про екземпляр базового протокола [1].

Опис середовища визначає сигнатуру базової мови та можливі обмеження інтерпретації цього опису (деяка частина підпису може бути інтерпретована на самому початку, для наприклад, числових функцій та предикатів, або конструкторів для станів агентів). Опис може також включати деякі конструктори для дій агентів, що занурюються (вставляються) у середовище. Набір базових протоколів визначає вимоги до поведінки системи та неявно визначає функцію занурення для даного середовища [18]. Вимоги формально можна виразити наступним чином: якщо передумова деякого створеного протоколу є

дійсною і процес цього протоколу розпочато, то після успішного завершення цього процесу постумова також є дійсною.

Семантику базових протоколів визначає різноманітність можливих реалізацій базових протоколів, які задовольняють формальним властивостям. На абстрактному рівні реалізація представлена як атрибутна система переходів, тобто розмічена система переходів з переходами які визначаються діями, станами та мітками атрибутів.

Інсерційне моделювання займається побудовою моделей та вивченням взаємодії агентів та середовищ у складних розподілених багатоагентних системах [21]. Неформально основні положення парадигми інсерційного моделювання можна сформулювати в таким чином:

1. Світ є ієрархія середовищ та агентів, занурених у ці середовища.

2. Агенти та середовища є сутностями, що розвиваються протягом часу та мають поведінку яку можна спостерігати.

3. Занурення агента в середовище змінює поведінку цього середовища та породжує нове середовище, яке готове до занурення в нього нових агентів (якщо для них є місце в цьому середовищі).

4. Середовище, яке розглядається як агент, може також бути занурене в середовище верхнього рівня.

5. Агенти можуть занурюватися в середовища середовищ верхнього рівня, а також породжуватись внутрішніми агентами, які вже є зануреними у середовище раніше.

6. Агенти та середовища можуть моделювати інші агенти та середовища на різних рівнях абстракції.

Під агентами та середовищами, маються на увазі як технічні, так і реальні системи – фізичні, біологічні та соціальні, а найбільш цікаві взаємодії – це насамперед інформаційні взаємодії, що абстраговані від фізичних процесів, які їх супроводжують [3]. При переході до математичних уточнень, поняття агента – це найбільш абстрактне математичне поняття, що моделює системи, які еволюціонують у часі. Таким чином, агент – це розмічена транзитивна система, стан якої визначається з точністю до бісимуляційної або трасової еквівалентності. Головною перевагою поняття транзитивної системи у порівнянні з іншими моделями систем, що еволюціонують у часі є поділ частини системи що спостерігається (виражається в діях), та прихованої частини системи, що визначається її внутрішніми станами [23].

$$< E, C, A, Ins > \tag{1.3}$$

Середовище - це агент, який має функцію занурення. Формально середовище має наступний вигляд (1.3), де *E* — множина станів середовища, *C* — множина дій середовища, *A* — множина дій агентів, які звнурюються у це середовище, *Ins* → функція занурення. Таким чином, будь-яке середовище *E* допускає занурення будь-якого агента з безліччю дій *A*. Оскільки стани транзитивних систем розглядаються з точністю до бісимуляційної еквівалентності, то їх можна ототожнювати з поведінками та говорити про безперервність функцій. Основна вимога до середовища – це безперервність функції занурення. З цього припущення випливає низка корисних наслідків. Наприклад, той факт, що функцію занурення можна визначити як найменшу нерухому точку системи функціональних рівнянь. Результат *Ins(e,u)* занурення агента, що знаходиться в стані u, в середовище, що знаходиться в стані e, позначається також як *e[u]*. Зважаючи на те, що середовище є агентом,

його можна занурювати в середовище верхнього рівня, розглядаючи багаторівневі середовища [14].

Атрибутні середовища. Для багатьох практичних додатків поняття середовища та агента є надто абстрактним, оскільки ігнорує структуру станів середовищ та агентів. Крім того, при переході в термінальний стан втрачається інформація про стан середовища, якщо вона має структуру. Всю необхідну інформацію можна, передавати через дії, але це часто виглядає неприродно і перевантажено. Для того, щоб вирішити цю проблему, було введено поняття атрибутної транзитивної системи. Атрибутні транзитивні системи відрізняються від розмічених тим, що вони мають мітки не тільки на переходах, а й на станах. Алгебра поведінки для атрибутних систем відрізняється тим, що поведінки можуть бути розмічені атрибутними мітками. Для цієї мети крім префіксингу вводиться ще одна операція, операція розмітки. Тепер вся необхідна інформація про стан середовища може передаватися через її розмітку. Зокрема тепер може бути багато термінальних констант, що відповідають успішному завершенню або глухому куту (deadlock), оскільки вони можуть мати різні розмітки. Атрибутні середовища будуються з урахуванням певної логічної бази. Ця база включає набір типів (цілі, речові, перераховані, символьні, поведінки та ін.), які інтерпретуються на певних областях даних, символи для позначення констант з цих областей, і набір типізованих функціональних та предикатних символів. Частина цих символів інтерпретується (наприклад, арифметичні операції та нерівності, рівність для всіх типів та ін.)[25]. Неінтерпретовані функціональні та предикатні символи називаються атрибути. Неінтерпретовані функціональні символи арності 0 називаються простими атрибутами,

решта – функціональними атрибутами (неінтерпретований предикатний символ розглядається як функціональний з бінарною областю значень). Функціональні символи використовуються для визначення структур даних, таких як масиви, списки, дерева.

Над логічною базою атрибутного середовища будується базова логічна мова. Зазвичай це мова першого порядку, можливо, з кванторами. За потреби можуть включати певні модальності темпоральної логіки. Атрибутним виразом називається простий атрибут або вираз, що складається з простих атрибурів. Якщо всі вирази є константами, то весь вираз називається константим. У загальному випадку ядро атрибутного середовища складається з формул базової мови. Атрибутні середовища поділяються на два класи: конкретні та символьні.

Найбільш відомі способи специфікації програм та систем (програмних та технічних) знаходяться в областях темпоральної, динамічної та деяких інших видів модальних логік [18]. Однак, у більшості випадків вони застосовуються тоді, коли вже відома досить докладна модель системи та вирішується проблема перевірки властивостей цієї системи, заданих у логічній формі (model checking).

Іншим способом опису вимог та специфікації систем є опис локальних поведінкових властивостей системи. У математичному вигляді йдеться про властивості відносин переходів транзитивної системи, а у разі, коли система представлена у вигляді композиції середовищ та агентів, йдеться про опис функції занурення. Кінцева поведінка системи називається процесом. Іого параметрами є передумова та постумова. Від параметрів можуть залежати як параметри, так і поведінка системи. Локальна властивість може

розглядатися як формула темпоральної логіки, що виражає той факт, що якщо (для відповідних значень параметрів) стан системи задовольняє визначеним умовам, то поведінка може бути ініційованою і після успішного завершення розмітки нового стану який задовольнятиме визначеним умовам Не важко простежити аналогію між трійками Хоару (формули динамічної логіки) та локальними властивостями систем. Інша аналогія – це продукції, що широко застосовуються при описі систем штучного інтелекту.

Абстракції. При роботі з великими системами, такими як телекомунікаційні системи, мережі типу Інтернет, багатопроцесорні системи з великою кількістю компонент, неможливо маніпулювати повними описами їх станів. Тому, принято замінювати стани таких систем абстрактними об'єктами різного роду. В інсерційному моделюванні, як абстракції великих систем використовуються атрибутні системи, стани яких розмічені формулами базової мови Якщо стани самі представлені формулами, вони визначаються як символьні атрибутні моделі [16]. Для вивчення співвідношень між абстрактними та конкретними моделями було введено поняття абстракції та конкретизації атрибутних транзитивних систем. Формально відношення абстракції може бути визначене наступним чином (1.4).

$$(s, s') \in \mathrm{Abs}\ \forall (a \in \mathbf{BL})((s \models a) \Rightarrow (s' \models a)) \quad (1.4)$$

Система *s* називається прямою абстракцією системи *s* ', а система *s* ' зворотньою конкретизацією системи *s*, якщо існує таке відношення, яке являє собою відношення моделювання.

Верифікація. Мова базових протоколів, реалізована в системі VRS, допускає використання атрибутів числових та символьних типів (вільні терми), масивів, списків та функціональних типів даних.

Дедуктивна система забезпечує доказ тверджень у теорії першого порядку, що є інтеграцію теорій цілочисельних та мовних лінійних нерівностей, що перелічуються типів даних, вільних (неінтерпретованих) функціональних символів та теорію черг. У дедуктивній системі успішно доводяться або спростовуються лише деякі класи формул, тому при верифікації іноді можуть бути отримані відмови для деяких проміжних запитів. Насправді такі відмови відбуваються досить рідко і здебільшого не впливають на отримання залишкового результату [22].

Як мову процесів використовують мову MSC з інсерційною семантикою. Система також допускає використання мови SDL та відповідних діаграмних уявлень мови UML.

Одна з типових технологій застосовується для перевірки повноти та несуперечності базових протоколів. Два базові протоколи називаються несуперечливими, якщо за будь-якої конкретизації цих протоколів їх передумови не можуть бути одночасно істинними. Суперечливість двох протоколів означає, що для деяких станів вибір протоколу, який застосовується в цих станах, відбувається недетермінованим чином. Цей недетермінізм може бути небажаним, і система повідомляє про суперечливість розробнику як попередження. Система базових протоколів називається повною, якщо для будь-якого конкретного стану існує хоча б один базовий протокол, що можна застосувати до цього стану [27]. Неповнота системи базових протоколів означає можливість тупикового стану (dead lock).

Якщо система несуперечлива та повна то перевірку закінчена. Якщо ж знайдено випадок суперечності чи неповноти і він не прийнятий розробником через те, що не зрозуміло, чи є відповідний

стан досяжним, виникає нове завдання – перевірка недосяжності умови, що виражає порушення вимоги несуперечності чи повноти. Оскільки недосяжність умови означає інваріантність її заперечення, то можна знову застосувати статичний аналіз, намагаючись довести відповідні властивості. Якщо це вдається, завдання вирішено, інакше можна намагатися посилити відповідні властивості або застосувати динамічну верифікацію, тобто. конкретну або символьну генерацію трас, намагаючись довести чи спростувати досяжність шуканих властивостей.

Для реалізації інсерційних моделей деякого класу використовується інтерпретатор моделей, який називають інсерційною машиною [5]. Інсерційна машина складається з трьох основних компонентів:

1. Модельний драйвер. Ця компонента керує рухом моделі по дереву її поведінки, обчислюючи переходи з поточного у новий стан.

2. "Анфолдер" поведінки агентів. Поточний стан моделі представляється у вигляді алгебраїчного вираження у розширеній алгебрі поведінки. Вхідна мова допускає використання рекурсивних визначень для опису поведінки агентів та структур даних для опису поведінки або стану середовища. "Анфолдер" поведінки використовує ці описи для того, щоб отримати розкладання стану.

3. Інтерактор середовища. Приводить стан середовища до нормальної форми, використовуючи опис функції занурення. Після приведення до нормальної формі драйверу моделі залишається тільки вибрати напрямок руху та здійснити перехід або або зупинитися.

Розрізняються два типи інсерційних машин: машини реального часу або інтерактивні машини та аналітичні інсерційні машини.

Машини реального часу працюють у реальному чи віртуальному середовищі, взаємодіючи із зовнішнім середовищем у реальному часі. Аналітичні машини призначені для аналізу моделей, дослідження їх властивостей, розв'язання задач на інсерційних моделях тощо.

Відповідно модельні драйвери також поділяються на інтерактивні та аналітичні. Інтерактивний драйвер після нормалізації стану повинен вибрати в точності один перехід та виконати його шляхом передачі своєї дії у зовнішнє середовище. Інтерактивна машина з погляду зовнішнього спостерігача функціонує як агент, занурений у зовнішнє середовище з функцією занурення, що визначає закони функціонування цього середовища. Зовнішнє середовище, наприклад, може змінити поведінковий префікс стану внутрішнього середовища відповідно до своєї функції занурення. Інтерактивний драйвер може бути організовано досить складно. Він може мати критерії успішного функціонування та працювати як інтелектуальна система, збираючи інформацію про минуле, створюючи модель зовнішнього середовища та покращуючи алгоритми прийняття рішень щодо вибору дій з метою підвищення рівня успішності свого функціонування. Крім того, інтерактивний драйвер може мати інтерфейс для обміну фізичними сигналами із зовнішнім середовищем (наприклад, прийом візуальної або акустичної інформації, інформації про стан у просторі тощо) [12].

Аналітична інсерційна машина, на противагу інтерактивній, може розглядати різні варіанти прийняття рішень про дії, повертаючись до точок вибір і розглядаючи різні шляхи в дереві поведінки системи. Модель системи може включати також і модель зовнішнього середовища для цієї системи. У загальному випадку аналітична машина

здійснює пошук станів, що володіють заданими властивостями (досяжність цільових станів) чи станів, у яких задані
властивості порушуються. Зовнішнє середовище аналітичної машини може бути представлене користувачем, який взаємодіє з машиною, формулюючи завдання та керуючи активністю машини. Аналітичні машини, збагачені логікою та дедуктивними засобами, що використовуються для символьного моделювання систем.

.

РОЗДІЛ 2. СУЧАСНІ ТЕХНОЛОГІЇ СТВОРЕННЯ ІНТЕРФЕЙСІВ КОРИСТУВАЧА В ІНТЕРНЕТ

2.1 JavaScript та web-framework React.js

JavaScript — динамічна, об'єктно-орієнтована прототипна мова програмування. Найчастіше використовується для створення сценаріїв роботи веб-застосунків, які надають можливість клієнтові (пристрої кінцевого користувача, що може бути як звичайним монітором настільного компьютера, так і мобільним пристроєм) власноруч динамічно взаємодіяти з користувачем, керувати браузером, асинхронно обмінюватися даними з сервером, змінювати структуру та зовнішній вигляд веб-застосунку [31]. Окрім прототипної, JavaScript також частково підтримує інші парадигми програмування (імперативну та частково функціональну) і деякі відповідні архітектурні властивості, зокрема: динамічну та слабку типізація, автоматичне керування пам'яттю, прототипне наслідування, функції як методи (поведінку) об'єктів одного класу.

React визначають як декларативну, ефективну і гнучку JavaScript-бібліотеку, яка призначена для створення інтерфейсів користувача. Вона дозволяє компонувати складні інтерфейси з невеликих окремих частин коду — “компонентів”. У загальному вигляді структура стандартного React – застосунка наведена на (рисунок 2.1). Відповідно “web-page” – власно клієнтська частина застосунку, “Action” та “Reducer” - частини застосунку що відповідають за бізнес логіку, та перетворення даних, та команд отриманих від клієнта, у той вигляд який буде принятним для сховища, позначеного на (рисунок 2.1) як “Store”. Бізнес логіка також

може взаємодіяти із сторонніми сервісами, що відображено як "giphy API".

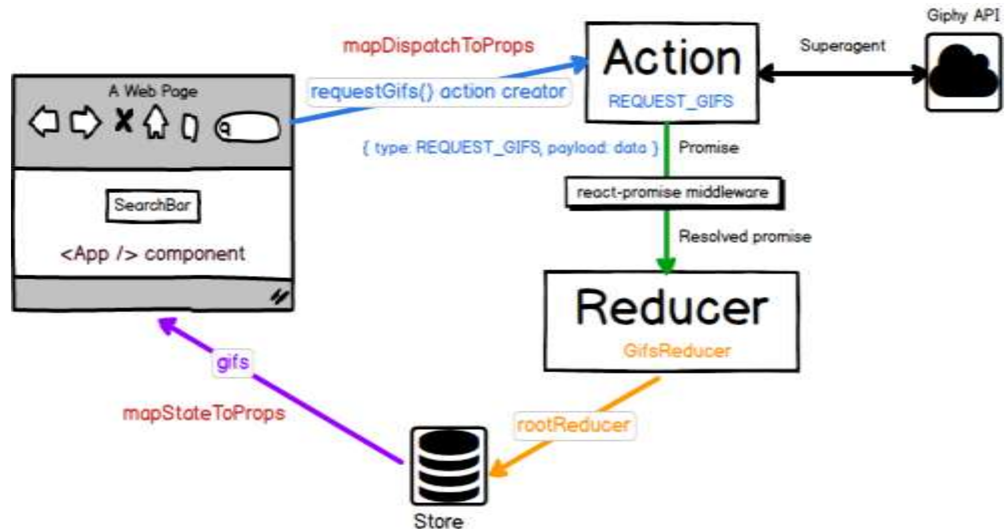

Рисунок 2.1. Загальна структура типового React-застосунка

ReactJS використовується як для розробки відносно "невеликих" застосунків (зазвичай до таких відносять інтернет застосунки у яких користувач працює самостійно, та не взаємодіє з іншими користувачами цього застосунку, як приклад можна навести різноманітні календарі, записні книжки, калькулятори, годинник та ін), так і для великих, корпоративних застосунків (систем які передбачають активну взаємодію користувачів між один одним, та складну ієрархію відносин між ними. Наприклад інформаційні системи керування учбовим процесом університету, або керування співробітниками великої корпорації). ReactJS надає мінімальний і надійний набір функцій для швидкого створення, налаштування та запуску веб-застосунків [29]. Можна визначити наступні головні особливості фреймворку ReactJS:

1. Одностороння передача даних — властивості (так у термінології React називають дані які зберігаються в екземплярах класів

або компонент, та мають певний рівень доступу) передаються від батьківських компонентів до дочірніх. Компоненти отримують властивості як безліч незмінних (англ. immutable) значень, тому компонент не може безпосередньо змінювати властивості, але може викликати зміни їх станів через функції зворотного виклику. Такий механізм називають «властивість униз, події наверх».

2. Віртуальний DOM. React використовує віртуальний DOM (англ. virtual DOM). React створює кеш-структуру в пам'яті, що дозволяє вирахувати різницю між попереднім і поточним станом інтерфейсу для плавного та своєчасного оновлення необхідної частини змісту сторінки у браузері. Таким чином програміст може працювати зі сторінкою, вважаючи, що вона оновлюється вся, але бібліотека самостійно вирішує, які компоненти сторінки необхідно перемалювати.

3. Технологія Redux - часто React використовують у зв'язку з Redux для управління станом компонентів

4. JavaScript XML (JSX) — це розширення синтаксису JavaScript, яке дозволяє використовувати HTML-подібний синтаксис для опису структури інтерфейсу. Як правило, компоненти React написані з використанням JSX. Також цілком допустимо використовувати звичайний JavaScript, проте слід пам’ятати що це призведе до значної втрати швидкості виконання сценаріїв на сторінці.

5. Не тільки малювання HTML в браузері - React використовується не просто для відображення готового HTML — шаблону у браузері, а також для заміщення певних частин сторінки відповідними компонентами. Наприклад, Facebook використовує. динамічну графіку, яка відображається в тезі <canvas> для програвання відео-кліпів. Netflix і PayPal використовують ізоморфні завантаження

для пошуку ідентичного HTML на сервері та клієнті, та заміни їх відповідними відео-програвачами. Тобто коли сторінка з великою кількістю відеороликів відправляється клієнтові, то спочатку вони всі представляють із себе звичайний текст або зображення. У той момент коли клієнт обирає елемент який його цікавить (натискає кнопку "програти" на відеопрогравачі), відео починає завантажуватись та поступово програватись у тому місті сторінки де раніше було зображення [25]. Коли або програвання відео буде зупинено користувачем, або коли закінчиться то об'єкт відеопрогравача буде знов динамічно замінений відповідним зображенням.

6. Хуки React - Хуки дозволяють використовувати стани вбудованих компонент та інші можливості React без необхідності написання класів. Побудова користувацьких хуків дозволяє повторно помістити логіку вже створеного компонента в інші функції, та використовувати методи або властивості компонента в процесі роботи функції.

Головні вимоги до компонентів в React полягають у тому, що вони повинні бути розроблені таким чином, щоб їх можна було легко передавати і інтегрувати з іншими компонентами. Також вони повинні розвиватися відповідно з передбачуваним життєвим циклом та підтримувати власний внутрішній стан. У якості основних складових частин типового застосунку на React виділяють наступні:

1. Компоненти. Це інкапсульовані блоки функціональності, які є основою React. Вони використовують дані (свойства і стан) для рендеринга користувацьких інтерфейсів (розглядаються, як компоненти React, що працюють з даними). Деякі їх типи також надають набір

методів контролю життєвого циклу, які можуть бути перехопленими іншими відповідними компонентами — перехоплювачами життєвого циклу. Процес рендеринга (виведення і оновлення інтерфейсу користувача на основі отриманих від сервера даних) в React є передбачуваним, і компоненти можуть підключатися до нього через відповідне React API.

2. Бібліотеки React. React містить набір важливих бібліотек. Головна бібліотека React працює з інтерактивними бібліотеками react-dom і react-native і орієнтована на специфікацію і визначення компонентів. Вона дозволяє створення компонентів дерева елементів сторінки, які можна використовувати для засобів візуалізації (рендерінгу) у браузері на іншій платформи(react-dom — один із таких засобів, призначених для відображення у браузері і на стороні сервера. Бібліотеки React Native сфокусовані на базових платформах і дозволяють створення React-застосунків для iOS, Android та інших платформ.

3. Сторонні бібліотеки. React не поставляється з інструментами моделювання даних, HTTP-викликів, бібліотеками стилів або іншими формами та елементами для інтерфейсу застосунка. Тому можливо використовувати у своєму додатку додатковий сторонній код, модулі чи інші інструменти. І хоча цими загальними технологіями React не комплектується, широка екосистема, оточуюча її, складається з неймовірно корисних бібліотек.

Однією із основних керуючих систем React - технологія віртуальної об'єктної моделі документа (віртуальна DOM, або vDOM).

Вона представляє собою структуру даних або набір структур даних, що імітують або відображають об'єктну модель документа, який використовується в браузерах. У цілому віртуальний DOM служить проміжним шаром між кодом застосунків і фактичним DOM-браузером.

У загальному вигляді схема роботи віртуального DOM відображена на (рисунок 2.2).

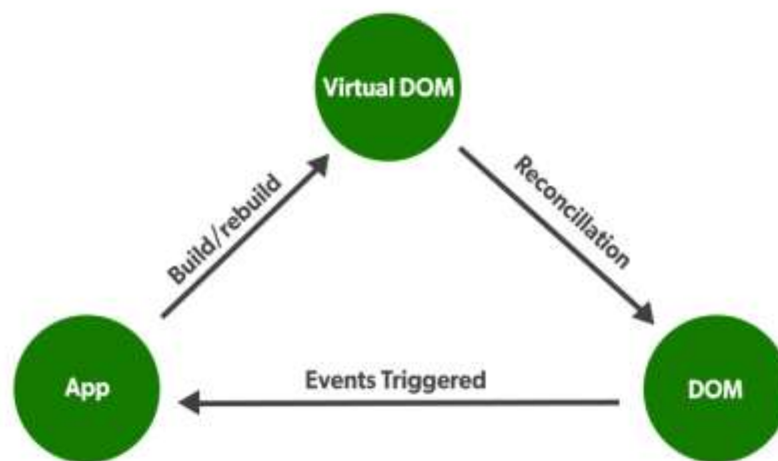

Рисунок 2.2. Загальний вигляд, роботи віртуального DOM

Як правило, віртуальний DOM дозволяє відокремити складність виявлення змін та управління від розробки та перейти до спеціального рівня абстракції [35].

DOM, або об'єктна модель документа (об'єктна модель документа), — це програмний інтерфейс, який дозволяє програмам JavaScript взаємодіяти з різними типами документів (HTML, XML і SVG). Для нього існують стандартизовані специфікації, тож публічна робоча група створила стандартний набір функцій, які повинні бути присутні в DOM, і варіанти поведінки моделі. Хоча існують і інші

реалізації об'єктної моделі документа, DOM в основному асоціюється з веб-браузерами, такими як Chrome, Firefox і Edge.

DOM забезпечує структурований спосіб доступу до документа, його зберігання та маніпулювання різними його частинами. В цілому DOM представляє собою деревоподібну структуру, яка відображає ієрархію XML-документа. Ця структура складається з піддерев, які, в свою чергу, складаються з вузлів (елементи div та інші, з яких сформовані веб-сторінки та застосунки). Частіше за все робота з DOM не викликає складностей, але ситуація може ускладнитися, якщо веб-застосунок великий, або необхідно у режимі реального часу відображати, або прибирати велику кількість вузлів, наприклад при візуалізації інтерактивного режиму системи інсерційного моделювання. У випадку прямої взаємодії із DOM, кожного разу коли система робить перехід до нового стану середовища, необхідно було б перезавантажити та перемалювати усе вже створене дерево поведінки [33]. Замість цього за відображення відповідає сама библиотека React. Інколи виникають ситуації, коли потрібно обійти віртуальну DOM і напряму спілкуватися з DOM.

Віртуальний DOM працює подібно до іншого світу програмного забезпечення — трьовимірними іграми. У таких іграх іноді використовується рендеринг, який працює приблизно так: отримати інформацію з ігрового сервера, відправити його в ігровий світ (візуальне представлення, яке бачить користувач), встановити, які зміни необхідно внести в цей світ, і тоді графична карта визначає необхідний мінімум змін. Одна з переваг цього підходу полягає в тому, що вам

потрібні тільки ресурси для роботи з поступовими змінами, внести які, як правило, набагато швидше, ніж оновлювати усю сторінку.

Це подібно до способу відображення та оновлення графічного зображення, що використовується у трьовимірних іграх. React намагається найбільш ефективним способом оновити інтерфейс користувача та використовувати методи, подібні до таких що застосовуються в трьовимірних іграх. React створює і підтримує віртуальну DOM в пам'яті як засіб відображення, обробляє оновлення DOM браузера на основі змін [32]. React може виконувати інтелектуальні оновлення і працювати тільки зі зміненими частинами даних, здатний використовувати евристичну звірку для вирахування того, які частини DOM в пам'яті вимагають змін у фактичному DOM.

React не тільки використовує новий підхід до роботи зі зміненням даних протягом часу; він також зосереджується на компонентах як на парадигмі для організації застосунку. Компоненти — основна одиниця React.

Компоненти React - інкапсульовані, багаторазові та складні. Ці характеристики забезпечують простий і елегантний спосіб проектування і створення користувацьких інтерфейсів. Застосунок складається з чітких стислих, логічних групп коду, якими досить просто керувати, та розвивати, додаючі нові функціональні можливості. Використання React для створення застосунків нагадує збірку моделей із деталей LEGO, з урахуванням того, що конструктор (програміст) має достатню кількість необхідних блоків і йому достатньо один раз створити певні компоненти [29], які у подальшому можливо використовувати безліч разів.

Добре спроектований компонент React повинен бути достатньо автономним, навіть якщо він використовує функціонал інших бібліотек, або навіть інкапсулює у собі певні їх частини. Розбиття користувацького інтерфейсу на компоненти, як правило, спрощує роботу з різними частинами застосунків, тому-що вносячи зміни лише у конкретну невелику частину застосунку, не обов’язково викликати повне пере-збирання усього застосунку. Розмежування компонентів означає, що функціональність і структура можуть бути чітко визначені, а їх автономність — що їх можна безліч разів використовувати і легко пересувати [40]. Компоненти в React також призначені для спільної роботи. Це означає, що їх можно зібрати разом для створення нових складових компонентів. Формування структури з компонентів — одна з найпотужніших можливостей бібліотеки React. Коли компонент створений один раз, він стає доступним для багаторазового застосування в іншій частині застосунку. Це особливо корисно у великих корпоративних застосунках.

До останньої області використання компонент у React відносяться методи життєвого циклу. Це передбачені надійні методи, які можуть діяти, якщо компонент знаходиться на різних стадіях свого життєвого циклу.

2.2. Компоненти React, та їх взаємодія

Компоненти — це фундаментальні одиниці клієнтського додатка, створені за допомогою бібліотеки React. Компоненти React з'єднані в деревовидні структури. Як і DOM-елементи, вони можуть бути вкладеними і містити інші компоненти. Засоби використання таких компонент дуже гнучкі. Зіставні компоненти часто легко пересувати і

використовувати багаторазово для створення інших компонент. Кожен такий компонент є самодостатнім, тому він може бути легко перенесений\скопійований в інші комплекти [27]. Переносимість — не найголовніша, але дуже ефективна особливість добре розроблених компонент React. Оскільки компоненти є зіставними, їх можна застосовувати в різних позиціях кода застосунку. Де б вони не використовувалися, вони допомагають сформувати певний тип відносин — батьківський або дочірній. Якщо один компонент містить інший, він вважається батьківським. А той, який знаходиться всередині другого, — дочірнім (нащадком). Компоненти, що знаходяться на одному рівні, не пов'язані між собою, та не можуть та взаємодіяти один і одним, хоча можуть бути розташовані поруч. Вони тільки «піклуються» про своїх батьків і нащадків.

Поряд з користувацькими методами і методами життєвого циклу класу, React надає стан (дані), яке можуть зберігатися разом з компонентом. Два основних типу станів — змінний і незмінний. Стан — це вся інформація, яка має доступ до застосунку в даний момент. Вона включає в себе, попри все, усі значення, які можуть бути передані до інших компонент, без будь-яких привласнень або обчислень у даний проміжок часу. До цієї інформації, наприклад, відносяться будь-які раніше створені змінні або інші доступні значення. Коли значення змінної змінюється, а не просто отримується її значення, змінюється стан програми — він більше не такий, яким був раніше. Можливо отримати стан в даний момент. Незмінні стани у React називають властивостями. Властивості — це дані, які передаються компонентам React або від батька, або від статичного методу defaultProps самого компонента. При цьому стан компонента локалізовано для одного

компонента, властивості зазвичай передаються із батьківського компонента.

Для роботи з властивостями існує кілька API: PropTypes і властивості за замовчуванням. PropTypes забезпечує функціональність перевірки типів, які можуть бути вказані як властивості компонента, і які застосунок буде очікувати в ході використання.

Також існує можливість використання функціональних компонентів, що не мають стану, записані як іменовані функції або виражені анонімні функції, призначені змінній [38]. Вони приймають тільки властивості і, оскільки повертають один і той самий результат на основі заданого введення, в основному вважаються чистими. Це обумовлює їх швидкість, так як React потенційно може виконати оптимізацію та уникнути непотрібних перевірок життєвого циклу або розподілу пам'яті. Функціональні компоненти без стану можуть бути потужними, особливо при використанні в поєднанні з батьківським компонентом, що має екземпляр підтримки. Замість того, щоб встановити стан для кількох компонентів, можна створити єдиний батьківський компонент, що має стан, і використовувати прості компоненти-нащадки для всього іншого. Також передача даних в React здійснюється лише в одному напрямку. Це означає, що замість горизонтального потоку між об'єктами, де кожен здатний оновлювати кожного, встановлюється ієрархія. Можливо також передавати дані через компоненти, але не можна виходити за рамки та змінювати стан або властивості інших компонентів без передачі властивості [37]. В окремих випадках існує можливість передавати дані вверх по ієрархії через зворотні виклики. Коли батьківський компонент отримує

зворотний виклик від компонента-нащадка, він здатний змінювати свої дані і відправляти змінені дані назад компонентам-нащадкам. Однак це використовується у дуже окремих випадках. Оскільки однонаправленный потік є особливо корисним при створенні інтерфейсів користувача, тому що спрощує представлення про те, як дані передаються по застосунку.

2.3. Життєвий цикл React застосунків

Методи життєвого циклу — це спеціальні методи, пов'язані з компонентами, заснованими на класі React, які будуть виконуватися в певних точках компонента життєвого циклу. Життєвий цикл — це повне відображення процесу існування компонента. Можна сказати що компонент з життєвим циклом має метафоричне «життя» — початок, середину і кінець. Ця “ментальна модель” прискорює розуміння функцій самого застосунку, показує в якому етапі свого життя він знаходиться, та найголовніше яку поведінку він зараз має. Так на “початку життя”, тобто при завантажені сторінки застосунок використовує методи для відображення свого змісту та елементів керування(кнопок, повзунків та ін.), після завершення роботи цих методів вони стають недоступними, та можна працювати лише з керівничими методами, так само наприкінці “життя” викликаються методи для звільнення пам’яті та ін [36]. . Методи життєвого циклу React не є унікальними — багато технологій користувацького інтерфейсу використовують їх через інтуїтивний характер і корисність. Основними частинами життя компонента React є ініціалізація, монтування, оновлення та розмонтування (рисунок 2.2).

Існують методи життєвого циклу, які будуть викликатися під час ініціалізації, а також до і після монтування, оновлення та розмонтування компонентів. Цих методів не так багато, особливо в порівнянні з іншими бібліотеками та фреймворками.

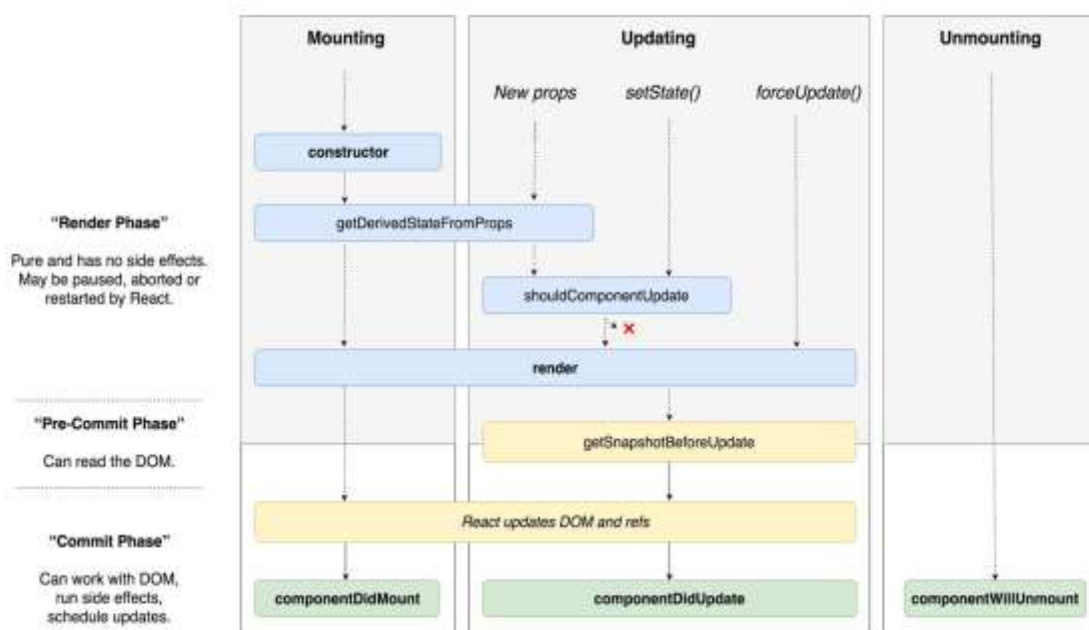

Рисунок 2.2. Загальна схема взаємодії компонент на основних етапах життєвого цикла React застосунка.

Монтування — це процес React, що поміщає компоненти у фактичний DOM. Після цього компонент отримує стан «готовий» і, як правило, настає саме час виконувати HTTP-виклики або читання cookie-файлів. На цьому етапі також можна отримати доступ до DOM-елемента через функцію ref,

Розмонтування — це процес видалення компонентів із DOM. Якщо застосунок написано за допомогою React, маршрутизатор видаляє компоненти при переміщенні по сторінкам.

2.4. Маршрутизація у React - застосунках

Маршрутизация — ключовий компонент усіх веб-сайтів і веб-застосунків. Вона грає центральну роль на простіших статичних HTML-сторінках і в найскладніших React-застосунках в Інтернеті. Маршрутизація вступає у гру практично кожен раз, коли необхідно додати URL-адресу з дією. Більшість застосунків заповнені URL-адресами, тому що посилання — це фактично засіб пересування мережею.

Раніше в базовій архітектурі веб-застосунків використовувався інший підхід до маршрутизації. Він включав сервер (створений на Python, Ruby або PHP), який генерував HTML-сторінку і відправляв її в клієнтові. Користувач міг натиснути на якусь кнопку, заповнити форму даними, відправити її назад на сервер і очікувати на відповідь.

З тої пори веб-служби притерпіли важливих змін у сфері проектування та побудови. Сучасні фреймворки JavaScript і браузерні технології достатньо розвинені для того, щоб веб-застосунки мали більш чітке розділення «клієнт — сервер». Сервер відправляє клієнтську програму (повністю браузерну), після чого застосовує свої механізми управління, повністю керуючи поведінкою застосунка. Також він відповідає за передачу користувачеві необроблених даних, зазвичай у форматі JSON.

Маршрутизація в сучасних клієнтських застосунках не вимагає перезавантаження всієї сторінки. Замість цього React може сам опрацювати весь процесс передачі даних на сервер, та отримання відповіді [25]. Це також дозволяє істотно зменшити час завантаження

застосунка у браузері і потенційно також оптимізувати навантаження на сервер.

2.5. Тестування компонентів React

На першому кроці тестування React-застосунка необхідно визначити яке тестове покриття буде найбільш ефективним для конретного застосунка. Найпростіша схема наведена на рисунку 2.3.

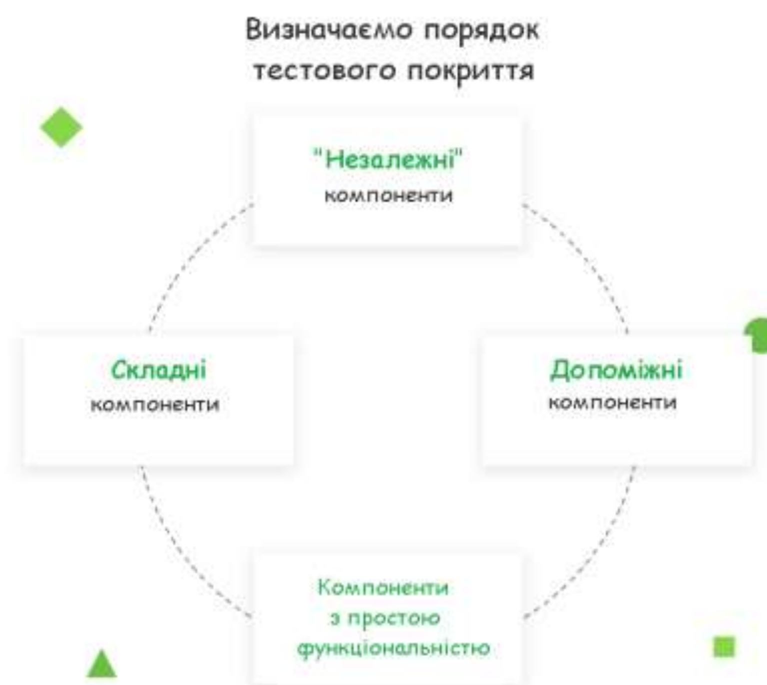

Рисунок 2.3. Визначення порядка тестового покриття React – застосунка.

Для тестування React-застосунка використовується кілька типів бібліотек.

1. Виконавець тестів — безпосередньо бібліотека яка керує процесом тестування. Більшість тестів можуть бути створені як звичайні файли JavaScript, але у більшості випадків виникає необхідність скористатися додатковими функціями виконавців тестів, наприклад запустити більше одного тесту за раз та повідомити про помилки або успішне завершення.

2. Дублювання тестів. При написанні тестів важливо щоб вони були якомога менше пов'язані з іншими чутливими до змін або непередбачуваними частинами інфраструктури (сторонніми бібліотеками, компонентами яки часто оновлюються); інші інструменти, повинні бути змодельованими тобто замінювати існуючи функції на «підробні» функції, які мають таку поведінку, яку від неї очікує розробник. Це допомагає зосередитись на тестованому коді та модульності, тому що тести не прив'язані до точної структури коду в даний момент часу.

3. Помічники з оточення. Тестування коду, який має запускатися у середовищі браузера, має дещо інші вимоги. Середовище браузера унікальне і включає DOM, події користувача та інші звичайні частини веб-застосунків. Розглянуті інструменти тестування допоможуть успішно відтворити поведінку та роботу браузера [36]. Частіше за все використовують бібліотеку “Enzyme” та “React test renderer” під час тестування компонентів React. Компонент Enzyme створений для прискорення процесу тестування. Він забезпечує надійний API, який дозволяє запитувати різні типи компонентів та елементів HTML, встановлювати та отримувати властивості компонентів, перевіряти та визначати стан компонентів та багато

іншого. React test renderer виконує аналогічні дії, а також може створювати моментальні знімки компонентів.

4. Бібліотеки, специфічні для конкретного фреймворку. Існують бібліотеки, створені спеціально для React (або інших фреймворків), які спрямовані на спрощення процесу написання тестів для певного фреймворку. Зазвичай їх розробляють для того, щоб оптимізувати процеси тестування бібліотек або всього фреймворку.

5. Засоби покриття. Завдяки детермінованому характеру коду можливо визначити те, які його частини покриваються тестами. Це дозволяє отримати чіткий показник, який відображає те наскільки повністю протестований код. Це не підміняє логіку та базовий аналіз (100%-ве охоплення коду не означає, що не можливе виникння помилок), але допомагає перевірити код.

Якщо узагальнювати підхід до тестування React-застосунку то можна виділити наступні принципи:

1. Тестування – це процес перевірки припущень про програмне забезпечення. Воно допомагає ефективніше планувати процес розробки компонент, запобігати збоям у майбутньому та підвищувати “впевненість” у коді. Також воно відіграє важливу роль у процесі швидкого внесення змін у застосунок на пізніх етапах життєвого циклу застосунка.

2. Різні типи тестів потрібно використовувати за різних обставин. Модульні тести, за своїм призначенням є найбільш поширеними та простими, дешевими та швидко створюваними. Інтеграційні тести перевіряють взаємодію багатьох різних частин системи, потребують значно більше ресурсів, та можуть бути

нестабільними, тому для їх написання та перевірки потрібно більше часу [24]. Зазвичай у великому застосунку таких тестів відносно небагато у порівнянні з модульними тестами.

РОЗДІЛ 3. СИСТЕМА ВІЗУАЛІЗАЦІЇ ГРАФІЧНОГО ПРЕДСТАВЛЕННЯ ПОВЕДІНКИ МОДЕЛІ У СИСТЕМІ ІНСЕРІЙНОГО МОДЕЛЮВАННЯ.

3.1. React застосунок для відображення поведінки моделі

Застосунок складається з двох основних частин. Клієнтська частина, позначена на (рис 3.1.) як React Application — застосунок, який має інтерфейс для забезпечення роботи із кінцевим користувачем, а саме надає можливість обрати необхідну конкретну інсерційну модель, яка потім передається до серверної частини на виконання.

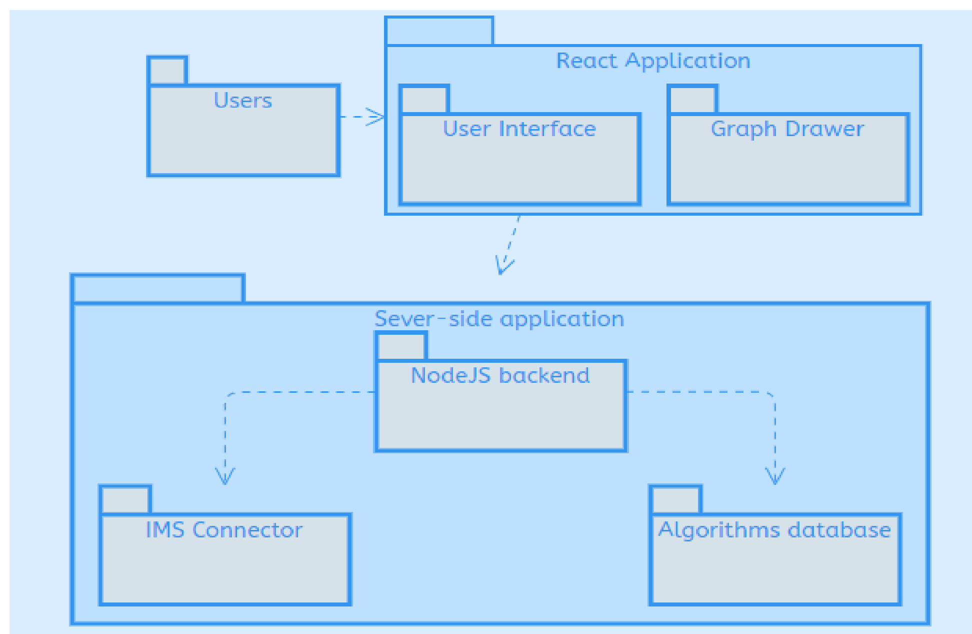

Рисунок 3.1. Архітектура застосунку

Далі керування бере на себе NodeJS-застосунок. Він завантажує обрану модель у вигляді JSON та передає її системі інсерційного моделювання для покрокового виконання. Система інсерційного моделювання має два режими роботи: автоматичний, та інтерактивний. В автоматичному режимі, ядро системи (Model Driver) обходить повністю все дерево станів моделі та виводить результат у файл. В

інтерактивному режимі система інсерційного моделювання робить один перехід, виводить поточний стан агента, можливі варіанти переходів до наступниї станів і очікує вибору користувача. Після того як користувач вказує той стан який його цікавить, система переходить до того стану. Саме цей режим найбільше підходить для інтерактивного відображення дерева станів в графічному застосунку. Оскільки, при вже існуючій великій кількості переходів з одного стану, у разі додавання наступного переходу до нового стану, може виникнути ситуація при якій буде необхідно перемалювати сусідні з ним стани.

Серверна частина складається власно з системи інсерційного моделювання, яка відповідає за виконання інсерційніх моделей та генерацію результатів роботи та модулю засованлму на фреймворку Node.js, який виконує роль “буфера” між клієнтською частиною і системою інсерційного моделювання, та відповідає за динамічне перетворення файлу який містить результати роботи моделі, отриманого від IMS у вигляді списку формату JSON. Цей модуль також бере на себе основну частину роботи по рендерінгу графічних примітивів (серверний рендерінг), які у клієнтському React – застосунку відображаются як вершини (стани агентів), та ребра (переходи між станами) дерева станів інсерційної маишини. Вершини графа позначаються мітками, які відповідають стану агента після здійснення системою інсерційного моделювання переходу в цей стан (). В наступній версії застосунка планується такоє додати вспливаючі вікна, які міститимуть повну інформацію про конкретний стан (рис. 3.2).

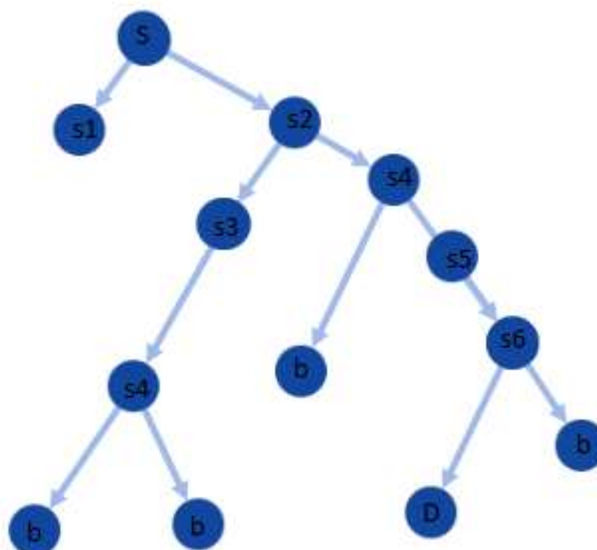

Рисунок 3.2. Вигляд дерева переходів

На наведеному на рисунку графі вершини позначені S та порядковим номером відповідають станам середовища після звичайного переходу (переходу з якого існують можливі шляхи подальшого перетворення системи, та який не є жодним із станів завершення). Вершини графа в даному випадку які позначені як *b*, означають стан, який в термінах інсерційного моделювання називається bot, стан який не є тупіковим, але з якого вже неможливі переходи (частіше за все цього стану можна досягти якщо перевищена максимальна припустима глибина дерева, що є одим із параметрів інсерційної машини), а вершина позначена як *D* – означає стан Delta, тобто успішне завершення процесу моделювання.

ВИСНОВКИ

В результаті виконання даної роботи було створено інтернет — застосунок, який забезпечує графічне відображення дерева станів та переходів інсерційної моделі в браузері, тоб-то на стороні клієнта. На стороні сервера створено серверний Node.js застосунок, який отримує обрану користувачем інсерційну модель, передає її на виконання системі інсерційного моделювання, виконує роль транслятора з формату виведення, що використовується системою інсерційного моделювання у формат JSON, та забезпечує передачу результатів роботи у вигляді JSON до React застосунку, який в свою чергу відповідає за відображення результатів на інтернет сторінці.

Напрямами подальшого дослідження за цією темою є зі сторони серверної частини — створення повноцінної бази даних, для коллекції вже існуючих алгоритмів, яка дозволить зберігати також історію роботи з конкретною моделлю, та робити статистичний аналіз результатів цієї роботи. Зі сторони клієнтської частини, подальше вдосконалення інтерфейса React застосунка, та механізмів взаємодії з інтерактивним режимом системи інсерційного моделювання дозволить розробити повнофункціональну систему “відлагодження” (англ. Debug) для інсерційних моделей.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, New York, 1985.
2. L. Aceto, Wan Fokking, and C. Verhoef. Structural operational semantics. In J. A. Bergstra, A. Ponce, and S. A. Smolka, editors, Handbook of Process Algebra, pages 197– 292. North-Holland, 2001.
3. Letichevsky and D. Gilbert. A Model for Interaction of Agents and Environments. In D. Bert, C. Choppy, P. Moses, editors. Recent Trends in Algebraic Development Techniques. Lecture Notes in Computer Science 1827, Springer, 1999.
4. Letichevsky, J. Kapitonova, A. Letichevsky Jr., V. Volkov, S. Baranov, V.Kotlyarov, T. Weigert. Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications. Computer Networks, 47, 2005, 662-675.
5. M.A. Reniers. Message Sequence Chart: Syntax and Semantics. Eindhoven, University of Technology, 1998.
6. International Telecommunications Union. Recommendation Z.120 Annex B: Formal semantics of Message Sequence Charts, 1998.
7. A.A Letichevsky, J.V. Kapitonova, V.P.Kotlyarov, A.A.Letichevsky Jr, V.A.Volkov. Semantics of timed MSC language, Kibernetika and System Analysis, 2002.
8. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^20 states and beyond. Information and Computation, 98 (2): 142–170, 1992.
9. T. Bultan and T. Yavuz-Kahveci. Action language verifier. In Proc. of ASE 2001, 382–386, November 2001.

10. S. Abramsky. A domain equation for bisimulation. Information and Computation, 92(2):161–218, 1991.

11. P. Aszel and N. Mendler. A final coalgebra theorem. In LNCS 389, pages 357–365. Springer-Verlag, 1989.

12. M. Roggenbach and M. Majster-Cederbaum. Towards a unified view of bisimulation: a comparative study. TCS, 238:81–130, 2000.

13. J. A. Goguen, S. W. Thetcher, E. G. Wagner, and J. B. Write. Initial algebra semantics and continuous algebras. J.ACM, (24):68–95, 1977.

14. A. Letichevsky. Algebras with approximation and recursive data structures. Kibernetika and System Analysis, (5):32–37, September-October 1987.

15. R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, oct 1991.

16. P. Azcel, J. Adamek, S. Milius, and J. Velebil. Infinite trees and comletely iterative theories: a coalgebraic view. TCS, 300(1-3):1–45, 2003.

17. A. Letichevsky and D. Gilbert. Interaction of agents and environments. In D. Bert and C. Choppy, editors, Resent trends in Algebraic Development technique, LNCS 1827. Springer-Verlag, 1999.

18. A. Letichevsky, J. Kapitonova, V. Kotlyarov, A. Letichevsky Jr, and V. Volkov. Semantics of timed msc language. Kibernetika and System Analysis, (4), 2002.

19. J. Rutten. Coalgebras and systems. TCS, 249, 2000.

20. R. Milner. A Calculus of Communicating Systems, volume 92 of Lecture Notes in Computer Science. Springer-Verlag, 1980.

21. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science, 96:73–155, 1992.

22. C. A. R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.

23. V. M. Glushkov and A. A. Letichevsky. Theory of algorithms and descrete processors. In J. T. Tou, editor, Advances in Information Systems Science, volume 1, pages 1–58. Plenum Press, 1969.

24. V. M. Glushkov. Automata theory and formal transformations of microprograms. Kibernetika, (5), 1965.

25. R. J. Glabbeek. The linear time — branching time spectrum i. the semantics of concrete, sequential processes. In J. A. Bergstra, A. Ponce, and S. A. Smolka, editors, Handbook of Process Algebra, pages 3–99. North-Holland, 2001.

26. P. Azcel, J. Adamek, S. Milius, and J. Velebil. Infinite trees and comletely iterative theories: a coalgebraic view. TCS, 300(1-3):1–45, 2003.

27. J. A. Bergstra, A. Ponce, and S. A. Smolka (Eds). Handbook of Process Algebra. North-Holland, 2001.

28. J. Rutten. Coalgebras and systems. TCS, 249, 2000.

29. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, Ed., Specification and Validation Methods. University Press, 1995, pp. 9–36.

30. J. Meseguer. Conditional rewriting logic as a unifiedmodel of concurrency. Theoretical Computer Science, 96: 73–155, 1992.

31. P. Lincoln, N. Marti-Oliet, and J. Meseguer. Specification, transformation and programming of concurrent systems in rewriting logic. In G. Bleloch et al., Eds., Proceedings of the DIMACS Workshop on Specification of ParallelAlgorithmsAmericanMathematical Society,

Providence, 1994.

32. M. Clavel. Reflection in General Logics and Rewriting Logic with Application to the Maude Language. Ph.D. thesis, University of Navarra, 1998.
33. M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kicrales, Ed., Reflection’96. 1996, pp. 263–288.
34. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.Quesada. Towards Maude 2.0. In F. Futatsugi, Ed., Proceedings of the 3rd InternationalWorkshop on Rewriting Logic and its Applications. Notes in Theoretical Computer Science, vol. 36, Elsevier, 2000.
35. J. Meseguer and P. Lincoln. Introduction in Maude. Technical report, SRI International, 1998.
36. J. Brackett. Software Requirements. Technical report SEI-CM-19-1.2, Software Engineering Institute, 1990.
37. J. A. Bergstra and J.W. Klop. Process algebra for synchronous communication. Information and Control, 60: 109–137, 1984.K.
38. Verschaeve, A. Ek., "Three Scenarios for Combining UML and SDL 96" Proceedings of SDL Forum'99, Montréal, Canada, June 1999.
39. Miyazaki, T. The complexity of McKay's canonical labeling algorithm // Groups and Computation, II, Amer. Math. Soc., Providence, RI, 1997. PP. 239-256
40. Spence E. Regular Two-Graphs on 36 Vertices // Linear Algebra and its Applications, 226-228 (1995) 459-497
41. Spence E. The Stromgly Regular (40, 12, 2,4) Graphs // The Electronical Journal Of Combinatorics. Vol 7(1), 2000.