Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут ім. І.Сікорського»
Навчально-науковий комплекс
«Інститут прикладного системного аналізу»

А. І. Петренко, Б. В. Булах

ПРИКЛАДНЕ ПРОГРАМУВАННЯ ЯК ОРКЕСТРУВАННЯ СЕРВІСІВ

Навчальний посібник

Призначений для магістерської підготовки за спеціальностями напрямків «Комп‘ютерні науки» і «Комп‘ютерна інженерія».

Київ 2016

УДК 004.75; 681.3

Петренко А.І., Булах Б.В.

Прикладне програмування як оркестрування сервісів. – Київ: НТУУ «КПІ», 2016. – 111 с.: іл.

Анотація:

Метою даного навчального посібника є ознайомлення студентів комп'ютерних спеціальностей з можливостями сервісно-орієнтованого підходу в розробці програмного забезпечення. Зокрема, розглядаються питання розробки веб-сервісів та організації їх узгодженого виконання. Посібник призначений в якості допоміжного матеріалу в рамках курсу «Сервісно-орієнтовані обчислення та архітектури» для студентів кафедри системного проектування ННК «ІПСА» НТУУ «КПІ», однак може бути корисним і для студентів інших кафедр, що здійснюють підготовку фахівців з комп'ютерних наук, та їх викладачам.

ЗМІСТ

Передмова ……5

Розділ 1. Базові засади сервісно-орієнтованих архітектур …… 6

1.1. Еволюція програмних архітектур …….. 6

1.2. Метрики оцінки розподілених архітектур …… 11

1.3. Основні положення сервісно-орієнтованої архітектури ……. 14

1.3.1. Історичні витоки СОА …….. 19

1.3.2. Переваги та недоліки СОА …… 21

1.4. Веб-сервіси як реалізація сервісно-орієнтованого підходу …… 23

1.4.1. SOAP-сервіси …….. 24

1.4.2. REST-сервіси …… 29

1.5. Розробка веб-сервісу на мові Java …….. 31

1.5.1. Платформи розробки веб-сервісів ……... 31

1.5.2. SOAP сервіс фільтрації зображень …….. 34

1.5.3. REST сервіс фільтрації зображень ……38

1.5.4. Завдання для самостійної роботи …….. 39

Висновки по розділу …….. 39

Контрольні питання …… 40

Розділ 2. Узгоджена взаємодія сервісів …… 41

2.1. Потоки робіт як модель багатокрокових обчислень ……. 41

2.2. Моделі та метрики потоків робіт …….. 44

2.3. Керування потоками робіт …… 55

2.3.1. Життєвий цикл потоку робіт …….. 55

2.3.2. Програмні засоби керування потоками робіт …….. 57

2.4. Потік робіт як сценарій взаємодії веб-сервісів …….. 66

2.4.1. Хореографія сервісів …….. 67

2.4.2. Оркестрування сервісів …….. 67

2.4.3. Композитний сервіс як бізнес-процес …….. 68

2.5. Розробка композитних сервісів на мові Java …….. 73

2.5.1. Композитний сервіс фільтрації зображення …….. 73
2.5.2. Бізнес-процес фільтрації зображення ……. 75
2.5.3. Завдання для самостійної роботи …….. 77
Висновки по розділу …….. 77
Контрольні питання ……. 78
Розділ 3. СОА і семантичні технології, грід та хмари …… 79
3.1. Ресєтрація сервісів …… 79
3.2. Семантичне розширення реєстру …….. 90
3.3. Грід-сервіси та їх оркестрування …….. 99
3.4. Хмарні сервіси …… 103
3.5. Розробка онтології сервісів …… 105
3.6. Розгортання сервісів у хмарі ……..... 106
3.7. Завдання для самостійної роботи ……. 107
Висновки по розділу …….. 108
Контрольні питання …….. 108
Перелік умовних позначень …… 109
Перелік літератури …… 110

Передмова

Сучасні наукові та інженерні застосування починають будуватися як складна мережа сервісів, запропонованих різними постачальниками, на основі гетерогенних ресурсів різних організаційних структур. Орієнтована на сервіси архітектура означає появу нової парадигми, яка є відповіддю на зростаючу складність та інтеграцію розподіленого програмного забезпечення. Іншими словами, йдеться про динамічну архітектуру програмного забезпечення, таку, де структура і поведінка програмного забезпечення міняється по ходу виконання завдань, так само як і місце розташування ресурсу, на якому відбувається функціонування даного програмного забезпечення.

Даний посібник має на меті ознайомлення читачів з можливостями сервісно-орієнтованого підходу в розробці програмного забезпечення. Зокрема, розглядаються питання розробки веб-сервісів та організації їх узгодженого виконання.

Розділ 1. Базові засади сервісно-орієнтованих архітектур

Даний розділ знайомить читача з базовими ідеями, покладеними в основу сервісно-орієнтованого підходу до побудови програмної архітектури розподіленої системи. Вже зараз багато сучасних наукові та інженерних програмних комплексів, а особливо --- бізнес-систем створюються як складна мережа сервісів, запропонованих різними постачальниками, на основі гетерогенних ресурсів різних організаційних структур. З якою метою? Тому, що орієнтована на сервіси архітектура є відповіддю на зростаючу складність програмного забезпечення, коли підтримка класичної монолітної системи, орієнтованої на індивідуальне використання, стає організаційно та комерційно невигідною. З іншого боку, зручність використання веб-інтерфейсу доступу до фунціоналу програмних пакетів, а також можливість використання віддалених обчислювальних потужностей для подолання обмеженості локальних ресурсів при виконання обчислень в адекватні часові терміни вже встигли оцінити багато користувачів.

1.1. Еволюція програмних архітектур

Під програмною архітектурою або ***архітектурою програмного забезпечення*** (ПЗ) будемо тут розуміти сукупність базових принципів та рішень, які мають визначальне значення для розробки програмного продукту, його функціонування та подальшої підтримки. Ми будемо зосереджуватись, в першу чергу, на принципах, що визначають декомпозицію функціоналу програмного продукту на складові блоки та подальшу їх композицію у єдине функціонуюче програмне забезпечення, звертаючи особливу увагу на взаємодію між складовими програмного комплексу. Говорячи простими словами, кожну програмну систему можна розглядати в розрізі її систем, підсистем та їх взаємодії.

Архітектура програмного забезпечення для прикладних обчислень розвивалася відповідно до розвитку технологій створення обчислювальних середовищ: від побудови системи з сильними зв'язками між компонентами навколо монолітного «математичного ядра», що максимально ефективно використовує обмежені ресурси ПЕОМ, до розподілених та грід-систем, що долають ресурсні обмеження перенесенням обчислень на інші машини у мережі. Досить умовно така еволюція зображена на рис.1.1.

Монолітна архітектура. Даний тип архітектури відрізняється тим, що функціональність підсистем обчислювального комплексу зосереджена у єдиній прикладній програмі, яка включає як прикладний функціонал, орієнтований на певну область застосування та вирішувані задачі (математичне ядро та підсистеми аналізу різних видів та рівнів складності і т.д.), так і забезпечуючі засоби (інтерфейс користувача, підсистеми побудови звітів, графіків, креслень тощо). Підсистеми є невід'ємними частинами у складі комплексу, який відрізняють сильні зв'язки між складовими.

> ***Примітка***. *«Силу зв'язків» між компонентами програмної системи можна «виміряти в цифрах». Про метрики виміру такого поняття, як «сила зв'язку між складовими» йдеться у наступному підрозділі посібника.*

Цей підхід має ряд переваг, таких як: мінімально можливі накладні витрати на взаємодію інтегрально сполучених підсистем, можливість їх найбільш ефективного поєднання для вирішення цільових задач та широке використання прийомів оптимізації програмного коду алгоритмів, що дозволяють підвищити швидкодію ПЗ та більш ефективно розпоряджатися ресурсами обчислювальної машини.

Недоліки такої архітектури очевидні: ПЗ з інтегрованим інтерфейсом користувача передбачене для використання безпосередньо на робочій станції користувача, а тому може використовувати ресурси з локальним доступом лише з цієї машини; крім того, сильні зв’язки між складовими ускладнюють процес розвитку та розширення функціональності.

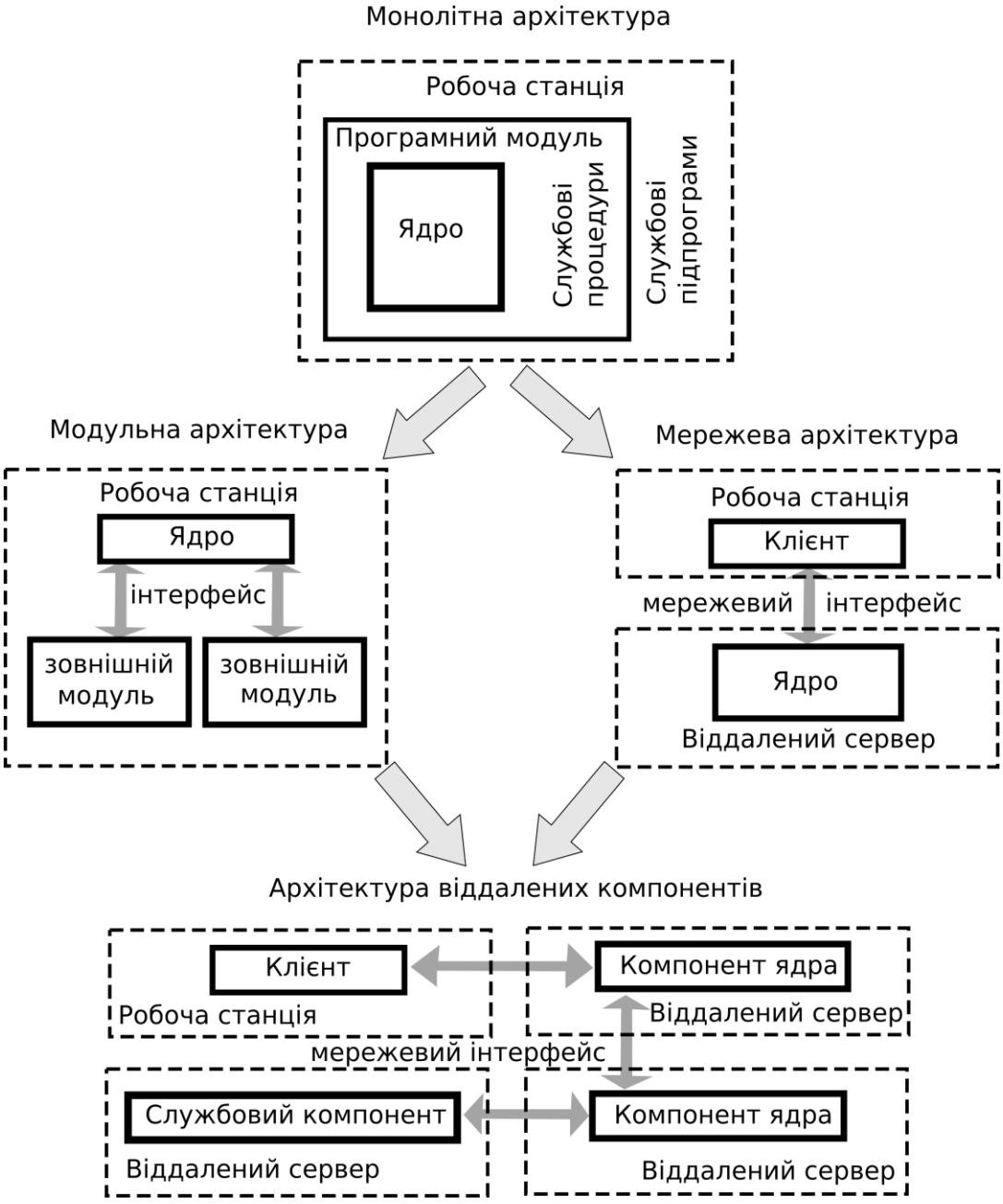

Рис.1.1 Шляхи еволюції програмних архітектур

Тим не менш, така модель архітектури залишається досить поширеною, наприклад, для САПР різноманітних вузьких спеціалізацій

(проектування механічних, електричних, гідравлічних систем та ін.). А от для міждисциплінарних комплексів прикладних досліджень така модель буде вже занадто обмежуючою.

Модульна архітектура. Подолання проблеми розширюваності ПЗ, як правило, відбувається шляхом послаблення внутрішніх зв'язків та підвищення ступеня модульності архітектури, що виражається в появі бібліотек описів функцій користувача, *бібліотек програмних компонентів користувача* (так званих «плагінів») та інших засобів розширення функціональності ПЗ без необхідності безпосереднього редагування програмного коду ядра.

Самі по собі компоненти розширення не вирішують інших проблем, таких як дефіцит обчислювальних ресурсів, окрім задачі спрощення процесу нарощування функціональності, в тому числі --- сторонніми розробниками. Для цього обов'язково визначаються та відкрито публікуються формати мов опису компонентів, їх даних, програмні протоколи та інтерфейси.

Мережева архітектура. Виходом з проблеми обмеженості ресурсів є розділення монолітного ПЗ на клієнтську інтерфейсну частину та серверне функціональне ядро, що дозволить перенести виконання функцій окремих підсистем комплексу на один або кілька високопродуктивних обчислювальних ресурсів, залишаючи при цьому користувача у звичному для нього робочому середовищі, що контактує з віддаленим ядром по мережі. Таке розбиття є прикладом застосування класичного ***клієнт-серверного архітектурного підходу*** в архітектурі ПЗ.

Клієнт-серверна декомпозиція вирішує важливу задачу *ізоляції функціоналу системи від інтерфейсу користувача*, спрощуючи весь процес розробки та супроводження системи та дозволяючи вільніше варіювати підходи до організації обчислень задля їх пришвидшення. До таких підходів, що успішно застосовані у комплексах інженерних розрахукнів, належать паралельні обчислення на багатоядерних процесорах та

багатопроцесорних системах, розподілені та грід-обчислення на мережі віддалених обчислювальних ресурсів.

Варто зазначити, що ефективні паралельні обчислення можливі за наявності вільних багатопроцесорних ресурсів, що не завжди є економічно прийнятним. Грід-обчислення пропонують використання об'єднаних вільних потужностей кількох організацій для виконання обчислень, що є перспективним підходом з точки зору показника "ефективність-ціна" (детальніше про грід-обчислення йдеться у третьому розділі). Однак безпосереднє розгортання комплексів прикладних обчислень в гетерогенному (як щодо апаратного забезпечення, так і програмного) середовищі грід потребує вирішення питань сумісності, інтеграції різних складових комплексу та ПЗ грід проміжного рівня (ПЗПР) на різних грід-ресурсах, поєднання грід та локальних обчислень, організації комфортної роботи користувача та ін., що вимагає подальшого пошуку кращих, більш універсальних архітектурних рішень.

Архітектура розподілених компонентів. З метою подолання проблем, притаманних попереднім архітектурним підходам, на увагу заслуговує підхід, що поєднує ідеї модульності та розподіленості, який можна охарактеризувати як «архітектура розподілених компонентів». Прикладом такого підходу, що здобув визнання при організації розподілених систем для різноманітних областей застосування, є ***сервісно-орієнтована архітектура (СОА, англ. Service-oriented architecture, SOA).***

Даний підхід передбачає подальшу декомпозицію функціональності на множину віддалених компонентів (сервісів) зі слабкими зв'язками, що можуть бути відносно незалежно розроблені та розгорнуті на наборі віддалених ресурсів, та поєднані у єдиний комплекс через єдині стандартні протоколи та інтерфейси. Загальними перевагами даного підходу для побудови комплексів наукових та інженерних розрахунків, систем автоматизації документообігу та інших прикладних застосувань є:

1. ***Слабкий зв'язок*** між компонентами-сервісами, притаманний модульній архітектурі, при стандартизованих інтерфейсах спрощує незалежну розробку та супроводження окремих компонентів, включаючи можливість залучення доступних компонентів від сторонніх розробників для розширення функціоналу системи.
2. ***Мережева природа*** компонентів-сервісів дозволяє їх розгортання на віддалених серверах, спрощує перенесення обчислень на вільні та/або високопродуктивні ресурси для подолання ресурсних обмежень. Взагалі, *сервіси виступають універсальним засобом організації доступу до унікальних віддалених програмних та апаратних ресурсів*.
3. ***Поєднання модульності та розподіленості*** дозволяє побудову системи, що за стандартними інтерфейсами компонентів приховує від користувача місце фактичного виконання їх програмного коду, разом з усіма низькорівневими деталями взаємодії з віддаленими ресурсами. При цьому стає можливим ***динамічне поєднання функціоналу сумісних за вхідними та вихідними даними компонентів-сервісів у складні послідовності виконання задля синтезу нової функціональності «на вимогу» відповідно до задач користувача***. Це ми називатимемо компонуванням сервісів.

1.2. Метрики оцінки розподілених архітектур

Вище часто згадувались поняття «сильний зв'язок», «слабкий зв'язок», які потребують додаткового уточнення. Основна ідея, викладена вище – слабкий зв'язок між компонентами ціною ефективності виконання обчислень спрощує розробку, підтримку програмної системи та забезпечує її гнучкість. Що ж таке сила зв'язку? Які є метрики виміру «складності» програмних компонентів та всієї архітектури? Про це йтиметься у даному підрозділі.

Пов’язність (cohesion). Ординальна величина, що зазвичай набуває значень «висока» чи «низька» (відповідно, вона придатна для порівняльної характеристики) --- міра того, як тісно і за якими принципами пов’язані частини одного модуля між собою. Розрізняють кілька видів пов’язаності:

- випадкова (нічого спільного),
- логічна (виконуються логічно подібні функції),
- часова (групування відповідно до моменту виконання),
- процедурна (дотримання певної спільної послідовності дій),
- комунікаційна (робота зі спільними даними),
- послідовна (вихід одної частини модуля є входом для іншої),
- функціональна (робота над спільним чітко визначеним завданням).

Ця характеристика є певним *мірилом адекватності компонентної декомпозиції* функціоналу програмного продукту: слабка пов’язаність (не плутати зі «слабкою зв’язністю» --- див. далі) коду одного модуля (компонента або сервісу) ускладнює його загальне розуміння та повторне використання. ***Висновок: окремі сервіси мають бути чітко спеціалізовані, виконувати чітко визначені функції в рамках однієї задачі.*** Цей принцип запозичений ще з об’єктно-орієнтованого підходу, а нині він же є основою і для мікросервісної архітектури.

Зв’язність (coupling). Ця міра (запропонував її Ларрі Константін ще у кінці 60х років минулого століття) показує ступінь залежності конкретного модуля від інших модулів програми. На відміну від попередньої метрики, ***перевагу мають слабкіші за зв’язністю рішення***. Так само тут виділяють кілька видів зв’язності (від «сильних» до «слабких»):

- за вмістом (залежність від внутрішнього стану інших модулів),
- за спільністю даних (спільні глобальні змінні),
- зовнішня (використання нав’язаного зовні формату даних),
- контролю (контроль ходу роботи інших модулів),

- структурна (використання різних частин спільної структури даних),
- даних (обмін спільними даними через атомарні параметри),
- через повідомлення (взаємодія через параметри та обмін повідомленнями),
- відсутня (взаємодія неможлива).

Природно, що намагання послабити взаємодію призводить до ускладнення архітектури. Чисельно зв'язність можна оцінити наступним чином:

$$Cp = 1 - \frac{1}{d_i + 2c_i + d_o + 2c_o + g_d + 2g_c + w + r}$$

де d_i --- кількість вхідних, а d_o --- кількість вихідних параметрів даних, c_i --- кількість вхідних, а c_o --- кількість вихідних керуючих параметрів (вищеназвані параметри описують зв'язність потоків даних та керування), g_d --- кількість глобальних змінних (даних), g_c --- кількість глобальних змінних для керування (ці параметри описують глобальну зв'язність), w --- кількість викликаних модулів, r --- кількість модулів, що викликають даний модуль (ці параметри описують зв'язність середовищ).

Зернистість (granularity). Дана характеристика є мірою рівня декомпозиції функціональності на модулі та окремі операції. Зернистість операції характеризує рівень складності виконуваних дій: арифметичні дії є прикладами дрібнозернистих «атомарних» операцій, а виконання частотного аналізу моделі електричного кола --- приклад крупнозернистої «супер-операції». Дану характеристику можна виразити лише відносно певної одиниці --- умовної базової операції:

$$G(a) = 1 - 1/n_a$$

де n_a --- кількість внутрішніх операцій, що віднесені до класу базових, при виконанні операції a потоку задач. Дана метрика сильно залежить від

вибору базової операції: так, якщо базовою операцією обрати просту арифметичну дію, то навіть прості матричні перетворення отримають низьку оцінку зернистості (тобто будуть «крупнозернистими»).

Інколи буває корисно розраховувати зернистість модуля (наприклад, веб-сервісу), що показує обсяг функціональності, який реалізується модулем. Оцінити можна аналогічно, однак враховуються всі операції модуля.

Оцінка зернистості важлива для дотримання балансу між малим числом «важких» модулів-сервісів, при якому гнучкість системи зовсім не виражена, та множиною простих сервісів, що забезпечують виконання базових дій і дозволяють істотно підвищити гнучність системи та її здатність до нарощення нового функціоналу на базі існуючого, однак згубно впливають на загальну ефективність через накладні витрати на взаємодію між атомарними сервісами.

Втім, важливо пам’ятати, що усі наведені метрики носять суб’єктивний, порівняльний характер і є лише допоміжним інструментом для прийняття проектного рішення щодо вибору кращої альтернативи з доступних варіантів сервісної декомпозиції.

1.3. Основні положення сервісно-орієнтованої архітектури

Сервіс-орієнтована архітектура (СОА, англ. SOA) – підхід до розробки програмного забезпечення, що покладається на використання інтероперабельних слабкозв’язаних компонентів, «сервісів», які мають узгоджені відкриті інтерфейси та використовують єдині правила (контракти) для визначення порядку взаємодії один з одним. В загальному ж розумінні під сервісом можна розуміти деякий програмний модуль із визначеною функціональністю та інтерфейсом. «Мережеве» поєднання сервісів (подібне до нейронних структур людського організму), в рамках

якого виходи одного сервісу можуть бути входами для іншого, дозволяє будувати системи зі складною поведінкою. Динамічне ж їх поєднання відкриває широкі перспективи для побудови автономних адаптивних систем. Надалі поняття СОА та сервісів буде конкретизовано.

Як вже згадувалося вище стосовно архітектури мережевих компонентів, СОА не виникла сама по собі як «ще одна» архітектурна модель для розподілених систем, вона є відповіддю на сучасні виклики в розробці програмного забезпечення. В даний час під тиском ринку швидко зростає складність розробки програмного забезпечення, і прагнення максимально пришвидшити цикл розробки вимагає пошуку нових архітектурних підходів.

Сучасні програми часто вже не є сталими, «консистентними» сутностями, як це було звично в минулому. Помітним є зміщення акцентів від монолітних програмних «ядер», що працюють на єдиній комп'ютерній платформі, до динамічно еволюціонуючих модулів. Інформаційні потоки також зазнають змін, і програми все активніше використовують обробку «онлайн-даних» з декількох, як правило, географічно розподілених джерел. Складові комплексних програмних продуктів створюються різними командами розробників за допомогою різних мов програмування. Такий «гібридний» підхід до розробки систем, звісно, ускладнює рішення задачі оптимізації продуктивності роботи складного програмного комплексу, підіймає на новий рівень проблему *інтероперабельності* (функціональної сумісності) його складових, однак часто є економічно обґрунтованим.

У підсумку ще раз підкреслимо, що потреба у створенні нового стилю розробки програмних додатків, основою якого є програмні сервіси (або служби, англ. software services), є обґрунтованою вимогами часу. Такий стиль дозволить програмістам не починати роботу з нуля, активніше спиратися на повторне використання коду: створювати нові програми, використовуючи вже готові розгорнуті сервіси, доступні в так званій

«екосистемі сервісів». Під екосистемою в даному контексті розуміється множина самих сервісів та набір зв'язків між ними: між сервісами існує певна ієрархія та розподіл обов'язків; обробка складного запиту веде до розбиття його сервісами верхніх ієрархічних рівнів на більш прості, які, у свою чергу обробляються спеціалізованими сервісами «екосистеми».

Рис.1.2 ілюструє можливі взаємозв'язки між сутностями подібної «екосистеми», які можуть належати як до сервісів (обслуговують запити), так і до клієнтів (надсилають запити-повідомлення). Пунктиром показані динамічні зв'язки, також показане середовище користувача (клієнт) та ресурси, з якими взаємодіють сервіси (формуючи високопродуктивні, хмарні та грід-сервіси). В подібній екосистемі можна виділити такі особливі підкласи:

- *сервіси-реєстри* (S_R, зберігають дані, необхідні для організації виклику інших сервісів),
- *композитні сервіси* (S_C, викликають інші сервіси, тобто є також самі по собі клієнтами інших сервісів),
- *сервіси-компонувальники* (або оркеструвальники, диригенти --- S_O, викликають довільні сервіси згідно отриманого *завдання на композицію*),
- *сервіси-агенти* (S_A, є «інтелектуальними» автономними сутностями, викликають довільні сервіси згідно власних «міркувань», для вирішення завдання, поставленого користувачем).

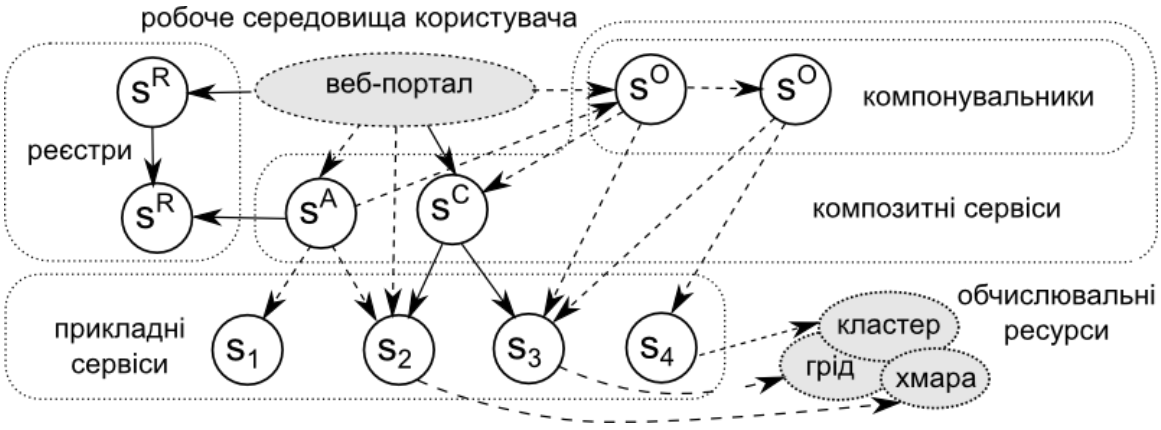

Рис.1.2. Приклад «екосистеми сервісів» комплексу прикладних обчислень

Нині як ніколи видається (в тому числі, з економічних міркувань), що жодна компанія або організація не ініціюватиме самостійного створення складних програмних додатків, не розглянувши можливості використання існуючих напрацювань. Для інтенсифікації повторного використання коду та даних, більшої гнучкості та масштабованості, навіть інженерні та наукові програми можуть будуватися як складна мережа служб, пропонованих різними постачальниками, на основі гетерогенних ресурсів під адмініструванням різних організаційних структур.

Тож враховуючи вищевикладені аспекти, не дивно, що концепції «сервісно-орієнтований комп'ютинг» (Service Oriented Computing --- SOC) і «сервісно-орієнтована архітектура» (Service Oriented Architecture --- SOA) протягом останніх років користуються стабільно високим інтересом, в тому числі у колах розробників систем автоматизованого проектування та систем організації інженерних розрахунків.

Згадаємо та уточнимо попереднє визначення програмної архітектури. *Архітектура - це формальний опис системи, що визначає цілі, функції, зовні видимі властивості та інтерфейси. Вона також включає опис внутрішніх компонентів системи та їх відносин, поряд з принципами, які визначають її дизайн, функціонування і можливу подальшу еволюцію.*

Конкретизуємо також поняття програмного сервісу в рамках розподіленої архітектури. *Сервіс (служба)* --- програмний компонент, до якого можна віддалено звернутися за допомогою комп'ютерної мережі, і який надає можливості з використання певного функціоналу запитуючій стороні (клієнту). Основна ідея SOA і SOC полягає в тому, щоб розподілити функціональність «крупнозернистих», монолітних додатків між сервісами, розгорнутими у мережі. Такі сервіси повинні бути здатні взаємодіяти між собою за допомогою стандартних інтерфейсів, що дозволяють їм прозоро працювати на базі різноманітних забезпечуючих платформ і між кордонами різних організацій.

Тобто, *можна охарактеризувати архітектуру SOA як слабкозв'язану, доступ до компонентів якої здійснюється через мережу відповідно до правил, чітко визначених цими компонентами*. Слід уточнити, що хоча у більшості випадків СОА асоціюється із використанням конкретно веб-сервісів, тим не менш, СОА можна реалізувати, використовуючи будь-яку технологію, засновану на вищеокресленій концепції служб. СОА позначає сам підхід до проектування систем, який спрямований на полегшення їх реалізації. Сервіс відповідає за реалізацію набору функцій з чітко визначеним інтерфейсом, включаючи опис шаблонів обміну повідомленнями, використовуваних при викликах функцій служби її клієнтами. Таким чином, використовуючи сервісно-орієнтовані архітектури, прагнуть досягти суворого поділу інтерфейсу і реалізації, необхідного для надання інших бажаних властивостей, наприклад таких, як інтероперабельність (здатність системи до взаємодії з іншими системами), прозорість розташування і слабка зв'язність між службами та клієнтом.

1.3.1. Історичні витоки СОА

Розробка подібних архітектур стала закономірним результатом еволюційного процесу, який розпочався ще у 80-ті роки з перших спроб побудови розподілених систем на основі об'єктно-орієнтованих моделей. Деякий час тому з'явилися розподілені системи наступного покоління, що базуються на компонентних моделях (напр. .Net, J2EE). Вони підсумували успадковані від попередників властивості, було впроваджено новий тип об'єктів, названих «компонентами». Таким чином, розподілені системи стали більш гнучкими, а також була забезпечена можливість виконання транзакційних операцій. Компонентна розподілена модель посіла належне місце у внутрішніх мережах підприємств, забезпечивши взаємодію гомогенних програмних додатків. Але розвиток технологій, в т.ч. Internet, проявив потенційні обмеження застосування таких платформ. По-перше, у кожній з моделей залучається власний комунікаційний протокол (RMI для J2EE, Net Remoting для .Net), і, хоча їх спільне використання можливе, воно часто неефективне. По-друге, занадто складна схема адресації; її застосуванню в середовищі Internet заважає необхідність централізованого присвоєння адрес компонентів. По-третє, структура повідомлень є низькорівневою. По суті, вона відповідає структурі, використовуваної в мовах програмування: ті ж цілі числа, символьні змінні, числа з плаваючою крапкою. При цьому не забезпечуються належна гнучкість, можливості додавання нових параметрів і самоопису об'єктів. Протоколи, по суті, синхронні, і компоненти все одно достатньо «сильно зв'язані», що суперечить окресленим вимогам будови великих систем. Відчувалася потреба в альтернативному, більш простому комунікаційному протоколі, і відповіддю на такі міркування можна вважати появу незалежного протоколу SOAP (Simple Object Access Protocol). В ньому використовується стандартна схема адресації з застосуванням URL для ідентифікації об'єктів,

транспортний протокол HTTP, дані в форматі ASCII і мова XML, що відкриває кращі можливості з самоопису об'єктів. Принципи СОА підтримуються мовою WSDL (Web Services Description Language), в якій опис сервісів ділиться на інтерфейс і його прив’язку до технічних деталей обміну повідомленнями. Інтерфейс описує, що саме повинні містити запит чи відповідь, а прив’язка визначає протоколи транспорту і даних.

Насправді, спроби реалізувати архітектуру додатків, в якій розподілені прикладні модулі представлені як об'єкти, що взаємодіють за допомогою чітко описаних інтерфейсів, робляться вже досить давно і з певним успіхом (моделі побудови компонентних додатків Common Object Request Broker Architecture (CORBA) і Microsoft Distributed Component Object Model (DCOM)). Зовні вони дійсно схожі на СОА, але більш детальний аналіз показує, що в цих ранніх підходах до сервісної орієнтації програмних систем не виконуються або виконуються з певними обмеженнями базові принципи СОА. Так, у CORBA і DCOM взаємодіючі об'єкти є «сильнозв’язними»: внесення змін у реалізацію програмних компонентів, що надають та використовують певний сервіс, повинні бути скоординовані між собою. Запити до об'єктів (сервісів) в різних архітектурах, як правило, містять невеликий обсяг інформації, яка враховує специфіку реалізації сервісів («дрібнозерниста» - англ. fine-grained - структура сервісів), і тому породжують значний мережевий трафік між постачальником і споживачем сервісу. У DCOM взаємодія програмних компонентів заснована на закритих інтерфейсах Microsoft. CORBA ж не належить приватній компанії, ця архітектура --- плід зусиль міжнародного консорціуму Object Management Group (скор. OMG, автор такого стандарту, як UML), який ставив за мету створення універсальної платформи інтеграції різнорідних програмних компонентів на базі стандартної мови опису інтерфейсів. Проте, реалізації специфікацій CORBA сильно варіюються в продуктах різних виробників, що обмежує інтероперабельність систем на

базі CORBA. Крім того, і CORBA, і DCOM мають суттєві обмеження щодо підтримки дійсно розподілених систем, їх протоколи взаємодії об'єктів занадто складні для організації продуктивного зв'язку сервісів, розгорнутих на різнорідних машинах.

1.3.2. Переваги та недоліки СОА

Основна перевага СОА полягає в тому, що така архітектура дозволяє будувати слабкозв'язані програми, тобто такі, в яких одну частину функціональності можна змінювати, не зачіпаючи інші. При цьому СОА ґрунтується на добре прописаних стандартах, а сервіси представлені так, що вони можуть бути легко відшукані (в реєстрах) і задіяні кінцевим користувачем або додатком.

Тож узагальнююче вищевикладене, серед головних принципів сервісно-орієнтованої архітектури, що обумовили її популярність, можна виділити наступні: максимальне повторне використання, модульність, здатність до поєднання, інтероперабельність (функціональна сумісність), відповідність стандартам, можливість ідентифікування, категоризації, моніторингу сервісів. Ці загальні принципи згодом конкретизувалися у наступний перелік:

1. **Стандартизований контракт**. Інтерфейс взаємодії сервісів описується документально та відкрито (у REST-сервісів інтерфейс описується лише для сприйняття людиною-розробником, а от у SOAP-сервісів опис інтерфейсу передбачає машинну обробку).
2. **Слабка зв’язність**. Взаємозв’язки між сервісами мають бути такими, що мінімізують взаємозалежності.
3. **Абстрагування сервісів**. Внутрішня логіка сервісу має бути прихована від зовнішнього світу, який обізнаний лише з його контрактом.

4. **Повторне використання**. Логіка розбивається на сервіси з думкою про вигоду повторного використання.
5. **Автономність сервісів**. Сервіси самі контролюють логіку, яку вони інкапсулюють.
6. **Відсутність внутрішнього стану**. Самі сервіси для кращої масштабованості і мінімізації використання ресурсів не повинні зберігати свій стан між викликами.
7. **Автоматизоване виявлення**. Сервіси супроводжуються метаданими, що уможливлюють їх автоматизоване виявлення та ідентифікацію.
8. **Здатність до компонування**. Сервіси мають бути добре придатними для поєднання задля синтезу нової функціональності «на вимогу».

Втім, не слід переоцінювати можливості СОА та вважати її панацеєю від проблем інтеграції та функціональної сумісності компонентів програмних систем. ***СОА є лише шаблоном***, за яким може бути збудована реальна інфраструктура, як успішна, так і невдала. Контракти сервісів вирішують питання сумісності лише на *синтаксичному рівні*, спираючись на стандарти опису інтерфейсів, однак для досягнення повної сумісності між споживачем сервісу та самим сервісом необхідна організація сумісності і на *семантичному рівні*, тобто забезпечення однозначного розуміння усіма учасниками множини використаних понять, позначень, для чого потрібні додаткові засоби. Без цього, наприклад, проблемним є організація автоматичного пошуку сервісів програмними агентами. Перехід на СОА також означає, як правило, додаткові накладні витрати на передачу повідомлень, виклик сервісів, а головне – змушує переносити традиційні програми на нові «рейки», що не завжди можливо та доцільно.

Однак розумне використання принципів СОА здатне полегшити розробку та підтримку розподілених програм, що спираються на численні

існуючі напрацювання, орієнтовані на розгортання на гетерогенних ресурсах.

В чому ж полягають можливі вигоди від використання СОА як архітектурного шаблону для таких програмних продуктів, як системи автоматизованого проектування чи комплекси моделювання? СОА полегшує розробку та підтримку таких продуктів за рахунок слабкої зв'язності компонентів-сервісів, розробку яких можна довірити різним групам розробників, оскільки стандартні контракти сервісів будуть запорукою їх сумісності. Реалізація функціоналу як набору сервісів дозволяє розгортати такі системи на кількох рознесених географічно різнорідних ресурсах (ПК, кластери, грід-ресурси тощо), а також оперативніше інтегруватися із сервісами сторонніх розробників. Розширення функціональності подібних комплексів можливе не лише за рахунок розробки нових сервісів, але й за рахунок нових сценаріїв їх взаємодії, ***створення композитних сервісів***. Втім, за подібну гнучкість та розширюваність доведеться заплатити накладними витратами, що неодмінно виникнуть внаслідок декомпозиції на сервіси та організації мережевої взаємодії.

1.4. Веб-сервіси як реалізація сервісно-орієнтованого підходу

На сьогодні саме специфікації веб-сервісів є найбільш перспективною та загальноприйнятою реалізацією принципів СОА, і терміни «веб-сервіс» та СОА часто помилково вважають синонімами. Однак самі веб-сервіси, як і СОА, в свою чергу також можуть бути реалізовані різними шляхами.

Останніми роками серед розробників ведеться активна дискусія на тему ідеологій побудови веб-сервісів. Нині головна увага приділена двом різним підходам, а саме: SOAP (Simple Object Access Protocol) та REST (Representational State Transfer). Ці підходи виражають різні погляди на

будову веб-сервісів. Основним (традиційним) підходом прийнято вважати саме перший – SOAP, адже він розроблявся та пропагується такими авторитетами IT-індустрії як Microsoft, IBM, Sun Microsystems та організаціями W3C, TAG (Technical Architecture Group) та WSAWG (Web Services Architecture Working Group). Однак все більше популярності здобуває і альтернативний REST-підхід, з «мінімумом специфікацій».

1.4.1. SOAP-сервіси

Під SOAP-сервісами прийнято розуміти зв'язку стандартів «SOAP-WSDL-UDDI» (рис.1.3). Коротко призначення цих складових можна описати так:

- UDDI – «виявлення ресурсу (сервісу)»;
- WSDL – «опис ресурсу (сервісу)»;
- SOAP – «доступ до ресурсу (сервісу)».

Протокол SOAP

SOAP (Simple Object Access Protocol) — це протокол обміну XML-повідомленнями. Встановлюючи правила на структуру повідомлень, придатні і для односторонньої комунікації, SOAP особливо корисний при клієнт-серверній взаємодії у RPC-стилі (запит-відповідь). Одною з вагомих переваг протоколу SOAP над, скажімо, CORBA, є те, що постачальник сервісу не зобов'язаний надавати скомпільовані клієнтські «заглушки» для усіх типів клієнтів (SOAP взагалі не прив'язаний до платформи чи мови програмування). По-друге, протокол SOAP є більш дружнім до «файрволів». Однак ці переваги досягаються за рахунок меншої продуктивності, спричиненої витратами на обробку XML-повідомлень.

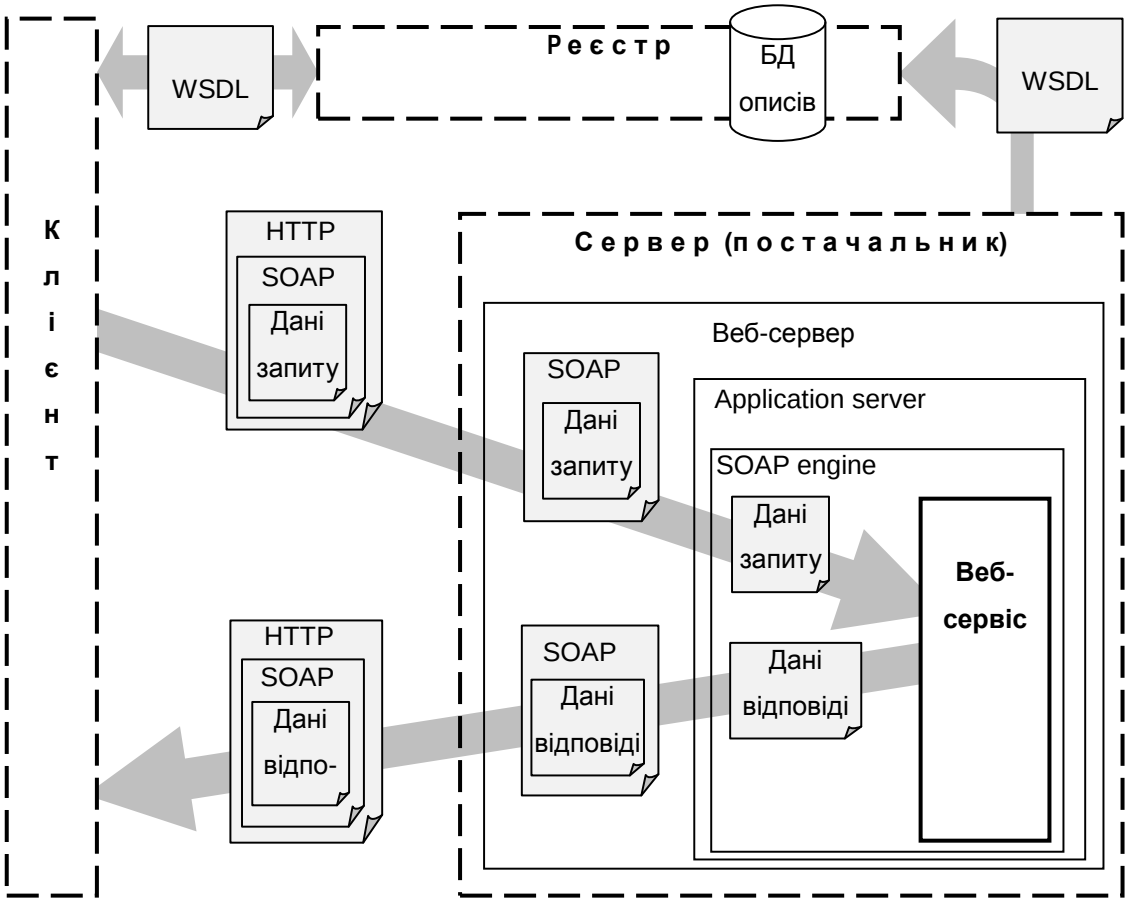

Рис.1.3. Архітектурні складові SOAP-сервісу

SOAP-повідомлення є синтаксично коректними (англ. well formed) XML-документами. Структурними елементами є «пролог» (необов'язковий елемент) з XML-декларацією та «SOAP-конверт» (або пакунок, англ. envelope) --- кореневий елемент, який містить елементи «заголовку» (необов'язковий) та «тіла» повідомлення. Характерними є дві речі: відносно велика частка службових символів (корисна інформація, фактично, полягає у назві методу, що викликається, назві параметрів та їх значеннях), та використання просторів імен (англ. namespaces). Використання просторів імен, визначених у XML-схемах, дає, з-поміж іншого, можливість застосовувати верифікацію повідомлень на відповідність XML-схемам, хоча це, звичайно, знизить швидкість обробки.

Тепер розглянемо послідовність взаємодії клієнта та сервера по протоколу SOAP. Почнемо з клієнта (рис.1.4). Вважатимемо, що для

транспорту повідомлень використовується (як це найчастіше і буває) HTTP. Це означає, що передаватися будуть SOAP-XML повідомлення, загорнуті у HTTP-запити. Те ж стосується і відповідей від сервера. З точки зору розробника програмного забезпечення, процес пакування-розпакування для зручності має бути реалізований в окремій бібліотеці (модулі), і такі бібліотеки є в наявності в достатній кількості для різних платформ.

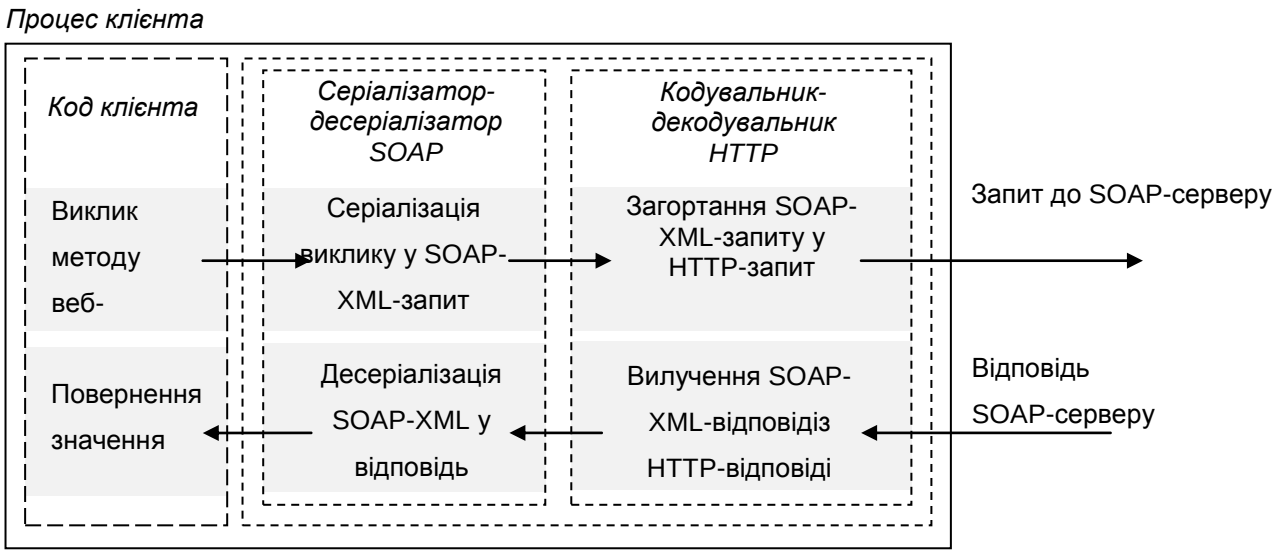

Рис.1.4. Взаємодія клієнтських програмних компонентів із сервером

При потребі викликати сервер, клієнтський код звертається до відповідного методу у SOAP-модулі. Цей модуль серіалізує виклик клієнта у SOAP-повідомлення і передає його у пакувальник HTTP-запитів. Готовий запит надсилається серверу, від якого надходить SOAP-HTTP-відповідь. Пакувальник HTTP-запитів розпаковує SOAP-повідомлення та передає його SOAP-(де)серіалізотору для десеріалізації. Десеріалізатор повертає виокремлене значення результату виконання запиту до клієнтського коду.

Реалізація серверного боку дещо складніша (рис.1.5): необхідний listener-процес («слухач») для відстеження вхідних повідомлень, та реалізація самого сервісу. Решта, тобто те, що стосується SOAP, лишається аналогічним до клієнтської сторони.

Вищезгаданий процес-прослуховувач зазвичай реалізується як сервлет (SOAP-сервлет) на веб-сервері. Ось, що відбувається. Процес на веб-сервері передає отриманий SOAP-HTTP-запит до SOAP-сервлету відповідно до вказаного в HTTP-запиті URL сервісу. Той, в свою чергу, використовуючи функціональність певної SOAP-бібліотеки з десеріалізації HTTP та SOAP-повідомлень, виокремлює параметри запиту (назва методу, параметри) та викликає реалізацію методу. Результат виконання методу загортається зворотнім шляхом у SOAP-конверт та HTTP-запит і передається веб-серверу для повернення до клієнта.

З вищевикладеного слідує, що протокол SOAP, з одного боку, --- простий та незалежний від ОС чи мови програмування, що є перевагою у випадку грід-середовищ. Він добре підходить під модель “запит-відповідь”, допускає механізми автоматизованої перевірки повідомлень на їх відповідність встановленим зразкам, в наявності є автоматизовані засоби для розробників програмного забезпечення (в т.ч. грід-сервісів). Однак, для використання, скажімо, у грід може знадобитися додатковий захист --- шифрування самого вмісту XML-повідомлень. Слід також відзначити чималі накладні витрати на пакування-розпакування повідомлень, істотно вищі ніж у інших протоколів типу CORBA. Це слід враховувати при розробці сервісів, які інтенсивно обмінюються значними об’ємами даних.

Мова опису інтерфейсу WSDL

WSDL (Web Service Description Language, мова опису веб-сервісів) як стандартна мова опису контрактів сервісів є важливою складовою технології веб-сервісів. Згідно підходу СОА, інтерфейси сервісів мають бути чітко описані та опубліковані. Ця вимога набуває особливого значення, коли в складній архітектурі мають поєднуватись сервіси різних розробників. WSDL-документ — це контракт веб-сервісів, мова WSDL — це мова опису інтерфейсів веб-сервісів. WSDL-опис є також XML-документом, що зручно

з точки зору наявних засобів парсингу для різних мов програмування. Існує дві специфікації WSDL – перевірена часом і досі популярна 1.1., та офіційно рекомендована W3C у 2007 р. нова версія 2.0. Розглянемо деякі особливості мови WSDL на прикладі версії 2.0, зважаючи на певні відмінності між версіями.

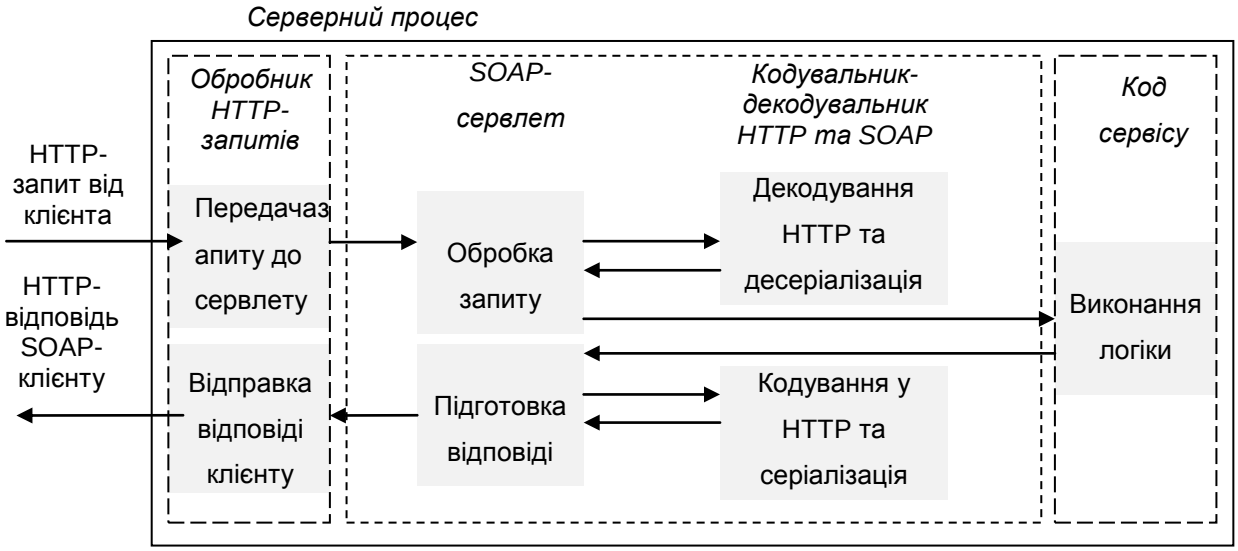

Рис.1.5 – Серверний процес взаємодії з SOAP-сервісом

Опис веб-сервісу можна розділити на дві частини. У «абстрактній частині» веб-сервіс описується типами повідомлень, які сервіс приймає та відправляє (зазвичай з використанням XML-схеми). Шаблони обміну повідомленнями визначають послідовність та кількість повідомлень при виклику методу (операції) сервісу. Елемент operation зв’язує шаблони обміну повідомленнями з одним або кількома типами повідомлень. Елемент interface (у версії 1.1 - portType) групує операції (елементи operation) незалежно від протоколу передачі даних. А вже прив’язка операцій до конкретних протоколів передачі та їх параметрів описується у секціях binding та service.

Ще раз підкреслимо значимість WSDL-контракту для побудови архітектури на веб-сервісах (включаючи т.зв. «грід-сервіси»): опис

інтерфейсу сприяє автоматизації при взаємодії з веб-сервісом не лише людини, розробника програмного забезпечення, а й інших сервісів, дозволяє автоматично компонувати сервіси згідно їх інтерфейсів у складні маршрути. Також наявність стандартних механізмів доставки повідомлень про помилки є досить корисною властивістю, хоча усі сценарії, закладені у шаблони обміну повідомленнями є синхронними. Асинхронні ж оповіщення не входять до стандарту WSDL 2.0. Також WSDL 2.0 все ще бракує семантики, що може бути виправлене за рахунок додаткових анотацій.

1.4.2. REST-сервіси

REST (англ. Representational State Transfer) --- це *набір архітектурних принципів і стиль проектування додатків*, орієнтований на створення мережевих систем, в основі яких лежать механізми для опису ресурсів та звернення до них. Прикладом такої системи може служити World Wide Web. Сам термін REST, запропонований Роєм Філдінгом у його дисертації, часто використовується у більш широкому сенсі для опису інфраструктури передачі даних через протоколи, подібні HTTP, без додавання додаткових семантичних шарів або спеціального управління сесіями.

У REST визначається суворий поділ відповідальності між компонентами клієнт-серверної системи, що полегшує реалізацію її основних «акторів». Іншою метою REST є спрощення семантики взаємодії компонентів мережевих систем, що дозволяє поліпшити масштабованість і підвищити продуктивність. В основу REST закладений принцип автономності запитів, що означає, що запити, оброблювані клієнтом або сервером, повинні включати всю контекстну інформацію, необхідну для їх розуміння. При роботі REST-систем для обміну даними стандартних медіа-типів використовується мінімальна кількість запитів.

REST-системи використовують URI --- універсальні ідентифікатори ресурсів для пошуку та отримання доступу до представлень необхідних ресурсів. Такі представлення, також звані репрезентативними станами, можна створювати, одержувати на запит, модифікувати і видаляти. Наприклад, принципи REST можна застосувати при реалізації системи публікації документів, щоб зробити їх доступними для читачів. У будь-який момент часу видавець може надати веб-посилання (URL), за допомогою якого читачі зможуть отримати доступ до інформації (репрезентативного стану) про опубліковані документи. Все, що потрібно читачам для доступу до інформації, що міститься в документах, --- це URL. Крім того, за наявності відповідних прав доступу, вони зможуть редагувати документи.

Одним з характерних принципів REST є те, що він дозволяє задіяти існуючі технології, стандарти та протоколи, створені для Веб, наприклад, HTTP. Будучи заснованим на використанні простих стандартних технологій і протоколів, REST виявляється простіше у вивченні та використанні, ніж більшість інших стандартів для обміну повідомленнями у Веб, тоді як організація ефективного обміну інформацією не вимагає серйозних додаткових витрат.

Обмін даними в стилі REST проходить без збереження стану, завдяки чому REST можна використовувати для спрощення процесу передачі інформації при використанні технологій, заснованих на підписці, наприклад RSS чи Atom, коли контент доставляється заздалегідь підписаним клієнтам.

В REST-системі, заснованій на HTTP, для доступу до репрезентативних станів ресурсів ***використовуються стандартні методи HTTP, такі як GET, PUT, POST і DELETE.***

1. **GET** --- використовується для передачі поточного репрезентативного стану ресурсу від видавця до споживача;
2. **PUT** --- використовується для передачі зміненого репрезентативного стану ресурсу від споживача видавцеві;

3. **POST** --- використовується для передачі нового репрезентативного стану ресурсу від споживача видавцеві;
4. **DELETE** --- використовується для передачі інформації, яка необхідна для модифікації ресурсу, коли репрезентативний стан вилучений.

Існує велика кількість бібліотек для реалізації REST-підходу для створення веб-сервісів. Серед відомих слід зазначити такі як Restlet, Jersey, Triaxrs, Jboss RESTEasy, Apache CXF.

1.5. Розробка веб-сервісу на мові Java

1.5.1. Платформи розробки веб-сервісів

Як вже згадувалось раніше, технологію веб-сервісів як і сервісно-орієнтовану архітектуру впроваджують та підтримують велика група провідних компаній. Кожна з них пропонує свої платформи та рішення для розробки, тестування та підтримки веб-сервісів. В цілому, вони подібні між собою, адже надають функціонал, підкріплений однаковими стандартами, однак все ж деякі відмінності існують.

Першу групу засобів, яку варто відзначити, складають здебільшого комерційні інтегровані середовища розробки, що підтримують цикл розробки веб-сервісів. До таких продуктів належать наступні.

Borland / CodeGear / Embarcadero Studio

Фірму Borland можна сміливо назвати піонером в області розробки засобів створення веб-сервісів для різних платформ. Щоправда, нині вони втратили свої позиції в області середовищ розробки. У 2006 році Borland заявила про намір згорнути свій бізнес по розробці IDE і прийняла рішення відділити Developer Tools Group у повністю окрему дочірню компанію CodeGear.

Середовища розробки, головним чином, підтримують мови програмування Delphi та C++. Так, ще Delphi 6 дозволяла створювати і використовувати SOAP і WSDL на платформі Windows, Borland Kylix - на платформі Linux, а JBuilder - на платформі Java.

Oracle JDeveloper

Oracle має два підходи до створення і використання веб-сервісів: по-перше, фірма пропонує програмну інфраструктуру, яку розробники можуть використовувати для створення веб-сервісів, а по-друге, розробляє і продає програмні продукти як веб-сервіси.

Більш того, Oracle використовує веб-сервіси для того, щоб розширити сферу впливу за межі ринку баз даних. Початком цього стала поява E-Business Suite - набору корпоративних продуктів, які можуть працювати через Internet.

Для розробки веб-сервісів Oracle пропонує J2EE-сумісне середовище Oracle JDeveloper (цей продукт доступний у вигляді безкоштовної версії). Для виконання веб-сервісів застосовується сервер додатків Oracle Application Server, а для створення додатків, що використовують дані, СУБД Oracle Database.

Microsoft Visual Studio + .NET Framework

Microsoft відіграє активну роль на ринку засобів створення та споживання веб-сервісів і спільно з IBM бере участь практично у всіх пов'язаних з цією технологіях новаціях. Практично немає жодного стандарту (починаючи зі стандарту мови XML), у прийнятті якого не була б помітна роль Microsoft. Прикладом зацікавленості Microsoft в лідерстві на ринку веб-сервісів може служити факт створення спільно з IBM у лютому 2002 року асоціації Web Services Interoperability Organization (WS-I).

В якості платформи для Web-сервісів Microsoft пропонує .NET Framework, що повністю підтримує реалізацію технологій веб-сервісів. Для розробки веб-сервісів та програм-клієнтів Microsoft пропонує Visual Studio -

-- візуальне середовище, що підтримує, серед інших, такі мови програмування, як C++ та C#, й інтегрується з існуючими серверами компанії.

IBM WebSphere Studio

Хоча IBM і не була в числі перших компаній, які сформулювали своє бачення Web-сервісів, вона фактично є лідером (як і Microsoft) з просування стандартів і технологій, пов'язаних з веб-сервісами. В даний час компанія IBM не тільки пропонує широкий спектр продуктів для створення і впровадження веб-сервісів (від WebSphere Suite до засобів хостингу Web-сервісів, підтримки Web-сервісів на рівні СУБД), але й має певну політику щодо розвитку самої концепції веб-сервісів і активно бере участь в її просуванні, співпрацюючи з іншим лідером --- Microsoft.

Говорячи про пропоновані IBM продуктах для створення і впровадження веб-сервісів, слід в першу чергу відзначити такі засоби, як WebSphere Studio та IBM Rational Application Developer для створення сервісів на мові Java, та сервер додатків WebSphere Application Server --- ключовий продукт лінійки WebSphere.

Eclipse / NetBeans

Окрему групу засобів розробки складають відкриті середовища розробки, такі як Eclipse IDE та Netbeans, що також надають розвинутий інтерфейс користувача для проектування та впровадження веб-сервісів. Здебільшого, вони базуються на відкритих бібліотеках та програмних засобах, таких як засоби Java J2EE, засоби проекту Apache (Axis2, ODE, jUDDI) тощо, що робить саме такі засоби пріоритетними для розробок. Слід відзначити, що підтримка веб-сервісів нині реалізована для багатьох мов програмування.

1.5.2. SOAP-сервіс фільтрації зображень

Уявимо таку гіпотетичну навчальну задачу. Нехай маємо реалізувати набір веб-сервісів (на мові Java), які б обробляли зображення – розмивали, інвертували, усереднювали 2 зображення. Ці сервіси потім зможемо організувати в композитний фільтр. Для простоти зображення будуть передаватися у кодуванні Base64, як DataURL.

Почнемо з реалізації базової функціональності. Створимо ієрархію класів для фільтрів. Нижче наводиться код цих простих фільтрів.

DataURLImageFilter.java:

```
package edu.imageservices.core;

public interface DataURLImageFilter {
      public DataURLImageFilter setData(String dataURL);
      public DataURLImageFilter applyFilter();
      public String getResult(String format);
}
```

DataURLImageBasicFilter.java:

```
package edu.imageservices.core;
import javax.xml.bind.DatatypeConverter;
import java.awt.image.BufferedImage;
import java.io.*;
import javax.imageio.ImageIO;
import edu.imageservices.core.DataURLImageFilter;

public abstract class DataURLImageBasicFilter implements DataURLImageFilter
{
      protected int[][] pixels;
      protected int width, height;

      @Override
      public DataURLImageFilter setData(String dataURL) {
            try {
                  byte[] data = DatatypeConverter.parseBase64Binary
                        (dataURL.substring(dataURL.indexOf(",") + 1));
                  BufferedImage image = ImageIO.read(new
ByteArrayInputStream(data));
                  width = image.getWidth();
                  height = image.getHeight();
                  pixels = new int[width][height];
                  for(int row = 0; row < height; row++) {
                        for(int col = 0; col < width; col++) {
```

```
                              pixels[col][row] = image.getRGB(col, row);
                        }
                  }
            } catch (IOException e) {}
            return this;
      }
      @Override
      public String getResult(String format) {
            BufferedImage image =
new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
            for(int row = 0; row < height; row++) {
                  for(int col = 0; col < width; col++) {
                        image.setRGB(col, row, pixels[col][row]);
                  }
            }
            String dataURL;
            try {
                  ByteArrayOutputStream data = new ByteArrayOutputStream();
                  ImageIO.write(image, format, data);
                  dataURL = "" +

      DatatypeConverter.printBase64Binary(data.toByteArray());
            } catch (IOException e) {
                  return "";
            }
            return dataURL;
      }
      @Override
      public DataURLImageFilter applyFilter() {
            this.filter();
            return this;
      }
      abstract protected void filter();
}
```

DataURLImageDummyFilter.java:

```
package edu.imageservices.core;
import edu.imageservices.core.DataURLImageBasicFilter;

public class DataURLImageDummyFilter extends DataURLImageBasicFilter {
      @Override
      protected void filter() {}
}
```

Ці класи є основою-заготовкою для класів реальних фільтрів:

DataURLImageBlurFilter.java:

```
package edu.imageservices.core;
import edu.imageservices.core.DataURLImageBasicFilter;

public class DataURLImageBlurFilter extends DataURLImageBasicFilter {
      @Override
```

```
	protected void filter() {
		for(int row = 0; row < height - 1; row++) {
			for(int col = 0; col < width - 1; col++) {
				pixels[col][row] =
					((pixels[col]  [row]   >> 2) & 0x3f3f3f) +
					((pixels[col+1][row]   >> 2) & 0x3f3f3f) +
					((pixels[col]  [row+1] >> 2) & 0x3f3f3f) +
					((pixels[col+1][row+1] >> 2) & 0x3f3f3f)
				;
			}
		}
	}
}
```

DataURLImageInvertFilter.java:

```
package edu.imageservices.core;
import edu.imageservices.core.DataURLImageBasicFilter;

public class DataURLImageInvertFilter extends DataURLImageBasicFilter {
	@Override
	protected void filter() {
		for(int row = 0; row < height; row++) {
			for(int col = 0; col < width; col++) {
				pixels[col][row] = ~pixels[col][row];
			}
		}
	}
}
```

DataURLImageMergeFilter.java:

```
package edu.imageservices.core;
import edu.imageservices.core.DataURLImageBasicFilter;
import edu.imageservices.core.DataURLImageDummyFilter;

public class DataURLImageMergeFilter extends DataURLImageBasicFilter {
	private DataURLImageDummyFilter temp;

	@Override
	public DataURLImageFilter setData(String dataURL) {
		if (this.width == 0) {
			super.setData(dataURL);
		} else {
			temp = new DataURLImageDummyFilter();
			temp.setData(dataURL);
		}
		return this;
	}
	@Override
	protected void filter() {
		if (temp == null) return;
		for(int row = 0; row < height; row++) {
			for(int col = 0; col < width; col++) {
				pixels[col][row] =
					((this.pixels[col][row] >> 1) & 0x7f7f7f) +
```

```
                                ((temp.pixels[col][row] >> 1) & 0x7f7f7f)
                        ;
                }
            }
        }
}
```

Тепер перейдемо до реалізації сервісу інвертування на основі вищенаведеної логіки. Опис сервісу складається з трьох файлів: публікатора (відповідає за запуск простого сервера, що викликає логіку фільтра), інтерфейсу сервісу та логіки (реалізації) сервісу.

DataURLImageInverFilterServicePublisher.java:

```
package edu.imageservices.soap.invert;
import javax.xml.ws.Endpoint;
import edu.imageservices.soap.invert.DataURLImageInvertFilterServiceImpl;

public class DataURLImageInvertFilterServicePublisher{
      public static void main(String[] args) {

      Endpoint.publish("http://0.0.0.0:8080/DataURLImageInvertFilterService"
,
                  new DataURLImageInvertFilterServiceImpl());
    }
}
```

DataURLImageInverFilterService.java:

```
package edu.imageservices.soap.invert;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.WebParam;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
import javax.jws.soap.SOAPBinding.Use;

@WebService
@SOAPBinding(style = Style.DOCUMENT, use = Use.LITERAL)
public interface DataURLImageInvertFilterService {
      @WebMethod String applyFilter
            (@WebParam(name = "dataURL")      String dataURL,
             @WebParam(name = "outputFormat") String outputFormat);
}
```

DataURLImageInverFilterServiceImpl.java:

```
package edu.imageservices.soap.invert;
import javax.jws.WebService;

@WebService(endpointInterface =
"edu.imageservices.soap.invert.DataURLImageInvertFilterService")
public class DataURLImageInvertFilterServiceImpl implements
DataURLImageInvertFilterService {
	@Override
	public String applyFilter(String dataURL, String outputFormat) {
		return new edu.imageservices.core.DataURLImageInvertFilter()
			.setData(dataURL)
			.applyFilter()
			.getResult(outputFormat);
	}
}
```

Після компіляції можемо запустити публікатор та направити йому вхідний запит. Вхідний малюнок та результат наведені на рис. 1.5.

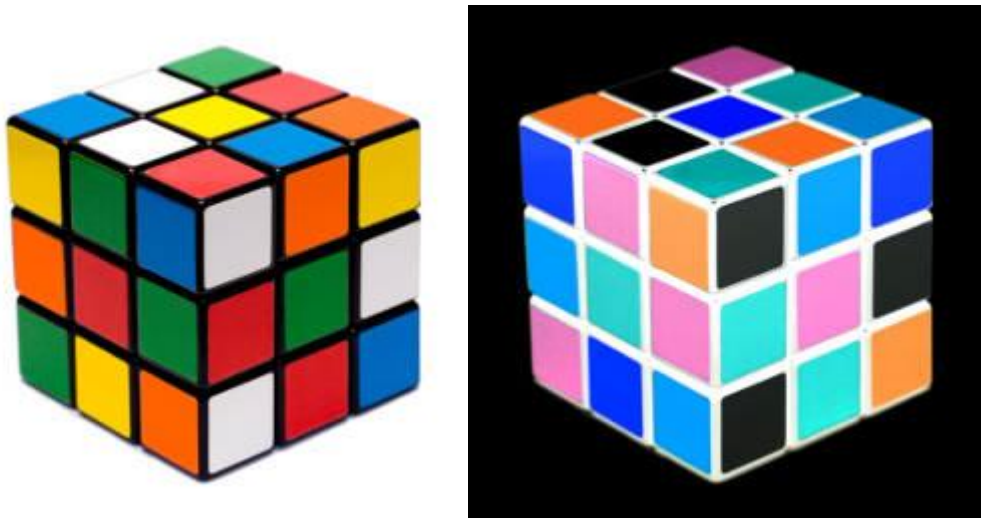

Рис. 1.5. Вхідні дані для сервісу-інвертора та результат його роботи

1.5.3. REST-сервіс фільтрації зображень

Для реалізації аналогычного REST-сервісу знадобиться залучити сервер Jersey + Grizzly. Клас ресурсу матиме вигляд:

```
package com.example;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
```

```
@Path("invertFilter")
public class DataURLImageInvertFilterService {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String applyFilter(String dataURL) {
        return new edu.imageservices.core.DataURLImageInvertFilter()
                  .setData(dataURL)
                  .applyFilter()
                  .getResult("png");
    }
}
```

1.5.4. Завдання для самостійної роботи

1. Спробуйте розрахувати окремі метрики для множини веб-сервісів з наведеного прикладу.

2. Спробуйте реалізувати самостійно повний набір REST-сервісів, аналогічно до того, як це було показано для SOAP-сервісів.

3. Спробуйте реалізувати набір сервісів для: обробки текстів (переклад, пошук ключових слів, виправлення помилок тощо), математичних операцій (вирішення систем лінійних алгебраїчних рівнянь тощо) або ін.

4. Познайомтеся з одним із вищезгаданих середовищ швидкої розроби, спробуйте розробити веб-сервіс, користуючись його (середовища) вбудованими засобами автоматизації розробки.

Висновки по розділу

1. Сервісно-орієнтована архітектура є відповіддю на сучасні вимоги до швидкого та ефективного проектування програмного забезпечення з використанням різних наявних засобів розробки різними колективами розробників з максимальним повторним використанням коду.

2. Сервісно-орієнтована архітектура є еволюцією модульної та мережевої моделей програмних архітектур, що поєднує їх сильні сторони.

3. Веб-сервіси є реалізацією концепції СОА, що здобула реальне визнання та прийняття в області розробки ПЗ.

4. Нині набули поширення два види веб-сервісів --- традиційні документовані SOAP-сервіси та «легкі» REST-сервіси.

Контрольні питання

1. Що таке СОА, які її переваги та недоліки?
2. Які є Вам відомі метрики оцінки розподілених архітектур?
3. Що таке веб-сервіс, його відношення до СОА?
4. Які є види веб-сервісів?
5. Опишіть процес взаємодії з SOAP-сервісом.
6. Які операції та формати даних підтримує REST-сервіс?
7. Що таке веб-сервер та контейнер сервісів?
8. Що таке контракт сервісу, його призначення?

Розділ 2. Узгоджена взаємодія сервісів

В даному розділі розглядається питання синтезу нової функціональності програмного продукту на основі ***сервісної композиції*** --- ***тобто узгодженого виконання веб-сервісів для досягнення заданої мети***. Однак розгляд цього питання ми проведемо крізь призму поняття потоків робіт або задач (англ. workflows). Тобто, моделлю організації обчислень виступатиме потік робіт, а реалізацією цієї моделі --- сервісна композиція.

2.1. Потоки робіт як модель багатокрокових обчислень

Під робочим потоком (або потоком / каскадом / маршрутом робіт) надалі в загальному випадку будемо розуміти множину кроків, кожен з яких представляє собою певну дію (обчислення), і переходів між ними, що організовують виконання цих кроків у певній послідовності.

Відповідно до того, як визначена семантика переходів, розрізняють такі підвиди робочого потоку, як *потік управління* та *потік даних* (рис.2.1). Реальний робочий потік є їх поєднанням.

Потік управління характеризує порядок виконання кроків потоку, та організовується за допомогою численних шаблонів, таких як *розгалуження, паралельне виконання, умовний перехід, цикл та ін.* (пригадайте блок-схему або UML Activity Diagram). Окрема увага може звертатися на обробку виключень та кроки з «компенсації / відкочування» (англ. rollback) виконаних дій (це дозволяє визначити дії з відміни багатокрокової транзакції в разі помилок на будь-якому її етапі). У потоці ж даних, що позначає шляхи передачі даних, головна увага відводиться саме питанням передачі даних: копіювання, область видимості, узгодження форматів даних тощо.

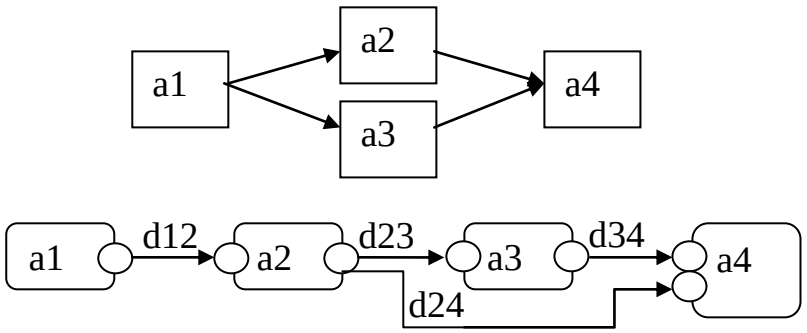

Рис.2.1. Підвиди робочих потоків: потік управління (зверху, перший крок грає роль розгалуження або умовного переходу) та потік даних (знизу, позначено порти вхідних-вихідних даних)

Формалізуємо вищеназвані концепції. Під конфігурацією потоку задач $w = (A_w,C_w,D_w)$, що належть множині допустимих конфігурацій потоків $w \in W$, розумітимемо сукупність:

а) множини $A_w \subseteq A$ обчислювальних операцій (дій) потоку $a_i \in A_w$, $i = 1..N$, з деякої множини доступних операцій A ($\forall a_i \in A_w, a_i \in A$);

б) множини $C_w \subseteq C$ переходів $c_{i,j} = (a_i,a_j) \in C_w$ між операціями з деякої множини допустимих переходів C, що задають послідовність виконання операцій в потоці (де $a_i \in A_w$ є операцією-попередником, $a_j \in A_w$ є операцією-наступником для даного переходу, що означає: операція a_j може бути виконана лише після a_i);

в) множини $D_w \subseteq D$ каналів передач даних $d_{i,j} = (a_i,a_j) \in D_w$ з деякої множини допустимих (сумісних за форматами) каналів D, що задають послідовність пересилання даних в потоці (де $a_i \in A_w$ є операцією-постачальником, $a_j \in A_w$ є операцією-споживачем даних, що означає: вихідні дані операції a_i потрібні для виконання a_j).

Інколи зручно розглядати потік задач окремо у розрізі послідовності виконання $w_C = (A_w,C_w)$ (потік керування) або у розрізі пересилок даних $w_D=(A_w,D_w)$ (потік даних).

Моделі потоку керування або потоку даних можуть бути відображені на звичайний направлений граф $G = (A_w,E)$, де у якості вершин виступатимуть операції потоку задач A_w, а у якості ребер E виступатимуть або переходи потоку керування C_w, або ж канали передач даних потоку даних D_w. Тобто ребра графа показують зв'язки-залежності між операціями-вершинами по порядку виконання або по даним відповідно. Як правило, залежності по даним автоматично визначають послідовність виконання (у іноземних джерелах це так званий «data-driven flow», тобто потік задач, керований даними), однак в загальному випадку графи C_w та D_w не обовязково співпадають, а тому доцільно відображати потік задач у вигляді одного графу з ребрами обох типів.

На рис. 2.2 наведено приклад графу типового потоку задач обробки даних зі сховища, в якому ребра переходів між операціями (суцільні лінії) та каналів даних (пунктир) не всюди співпадають: видалення непотрібних даних зі сховища (a_n+1) має бути виконано лише після того, як дані операції-постачальника a_0 будуть спожиті усіма операціями-споживачами.

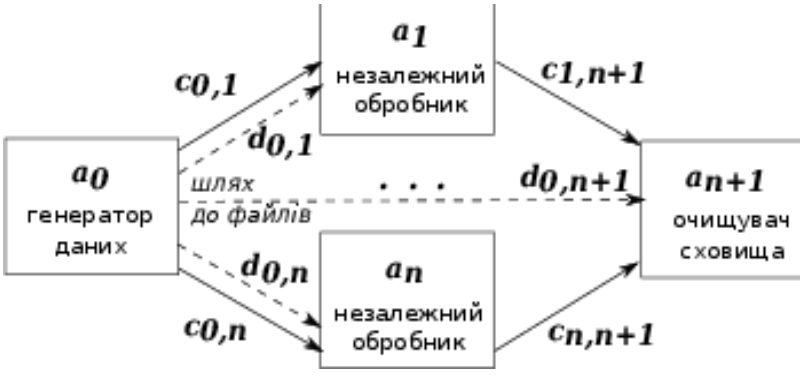

Рис. 2.2. Фрагмент потоку задач обробки даних у сховищі

Слід зазначити, що дана загальна модель описує саме обчислювальні потоки задач, пов'язані з обробкою інформації, що є типовими для віртуальних числових експериментів та систем комп'ютерного моделювання.

Використання формалізованих моделей потоку задач дозволяє реалізувати (у порядку важливості):

- автоматичне виконання потоків задач;
- контроль здійсненності потоку задач;
- пошук критичних “вузьких” ділянок у потоці задач;
- апріорну оцінку часу виконання потоку задач;
- оптимальне планування виконання потоків задач.

Загальна описова модель потоку задач $w = (A_w, C_w, D_w)$, що легко представляється орієнтованим графом, є зручною та зрозумілою абстракцією для користувачів. Втім, для вирішення задач моделювання дана модель не є найбільш підходящою, оскільки вона об’єднує операції зв’язками з різною семантикою, послідовністю виконання і каналами даних, і їй притаманний зависокий рівень абстракції для потоку даних.

2.2. Моделі та метрики потоків робіт

Для аналізу властивостей потоку робіт доцільно застосувати інші представлення та моделі, зводячи задачу моделювання потоків задач до добревідомих та добре досліджених методів дослідження комплексу пов’язаних задач. Такими моделями можуть слугувати мережеві графіки, різновиди мереж Петрі, опис за допомогою алгебр (числення) процесів тощо.

Час виконання потоку задач

Час виконання може вважатися основною об’єктивною метрикою для потоків задач, що дозволяє:

- проінформувати користувача стосовно очікуваної тривалості виконання його потоків задач;

- порівняти різні варіанти конфігурацій потоків задач між собою за критерієм мінімального часу виконання;
- порівняти різні варіанти реалізації елементів *систем керування потоками робіт (СКП)* за критерієм мінімального часу виконання еталонного потоку задач.

Оцінку часу виконання потоку задач та пошук вузьких місць може бути здійснено, зокрема, за допомогою методик мережевого планування. Мережевий графік представляє собою модель виробничого процесу, що враховує його внутрішні технологічні залежності та порядок виконання у часі складових операцій через введення понять «роботи» і «події». Робота відображає певний трудовий процес, виконання якого потребує певного часу та ресурсів. Подія позначає момент завершення певних задач та (або) початку нових. Графічно традиційно мережевий графік зображається у вигляді графа, вершини якого позначають події, а ребра --- роботи.

Зважаючи на подібність такого представлення виробничого процесу із моделлю обчислювального потоку задач (сировиною та продукцією, що виробляється, є дані, а ресурси представлені обчислювальними засобами), можна прямо застосувати методики розрахунку мережевого графіку та визначення критичного шляху для оцінки часу виконання потоку задач та пошуку його критичних ділянок.

Однак перед тим, як переходити до цієї або інших моделей оцінки часу виконання, слід ліквідувати неоднорідність зв’язків між операціями потоку задач, а саме множини переходів C_w та каналів передачі даних D_w. Цю задачу можна вирішити методом заміни D_w на «псевдо-операції» передачі даних A_D, тобто перетворення $(A_w,C_w,D_w) \rightarrow (A_w \cup A_D,C_w)$ як це проілюстровано на рис. 2.3 для прикладу потоку задач з рис. 2.2.

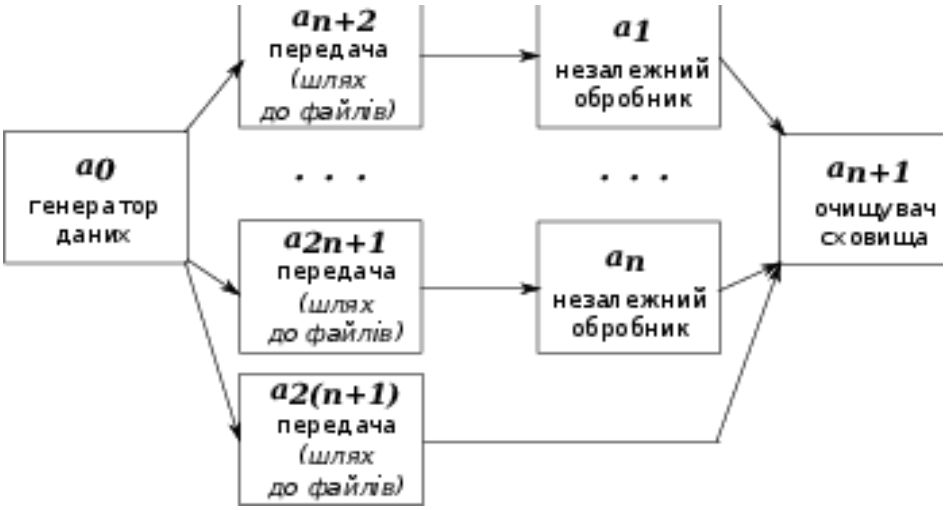

Рис. 2.3. Канали передач даних як псевдо-операції

Після виключення окремого класу ребер D_w, граф потоку задач прямо відображається мережевим графіком на рис. 2.4, незаповнені сектори --- для часових оцінок.

Розрахунок мережевого графіку, побудованого для потоку задач, потребує інформацію про час виконання окремих операцій потоку a_i і дозволяє:

- розрахувати мінімальний час виконання потоку задач;
- визначити перелік операцій, що визначають цей час (критичний шлях);
- відшукати резерви по часу, що разом з попередніми пунктами є базою для проведення оптимізації конфігурації потоку задач.

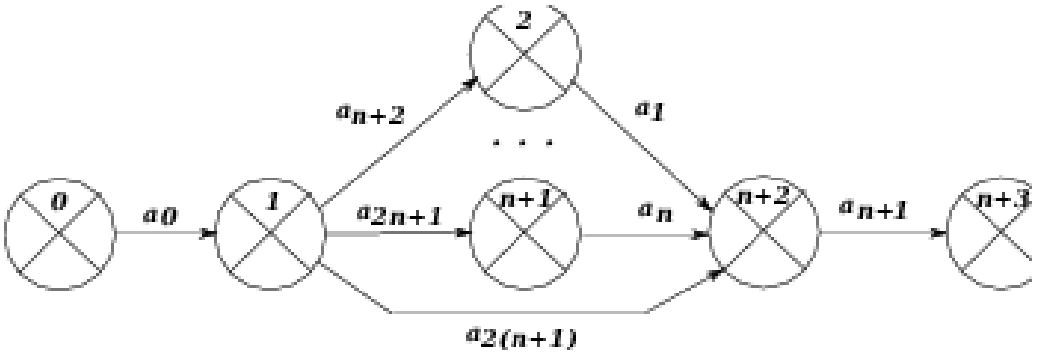

Рис. 2.4. Мережевий графік потоку задач обробки даних

Між тим, дана методика має певні недоліки, які перешкоджають її безпосередньому застосуванню для апріорного аналізу обчислювальних потоків задач маршрутів проектування:

- недопустимість циклів, що робить придатною лише для оцінки потоків задач, графи яких є ациклічними (англ. DAG);
- недопустимість паралельно виконуваних задач, що вимагає впровадження додаткових фіктивних задач;
- вважається, що виконання задач, що виходять з вершини-події, відбувається лише при виконанні усіх задач, які входять до неї («схема І»).

Альтернативним варіантом аналізу конфігурації потоку задач та обчислення часу його виконання є застосування моделей, що володіють достатньою виразною потужністю при можливості опису систем на різних рівнях абстракції та придатні для машинного імітаційного моделювання, таких як мережі Петрі, зокрема --- часові мережі Петрі.

Дослідження опису потоку задач часовою мережею Петрі (тобто такою, для переходів якої визначена вага, що визначає часову затримку спрацювання переходу) дозволяє:

- здійснити моделювання виконання потоку задач, що може містити цикли;
- здійснити моделювання виконання потоку задач, в якому затримки є випадковими величинами (за допомогою стохастичних мереж Петрі);
- дослідити потік задач на досяжність результату, наявність зациклювань та тупикових ситуацій.

Представлення наведеного вище прикладу (рис. 2.2) потоку задач мережею Петрі проілюстровано на рис. 2.5.

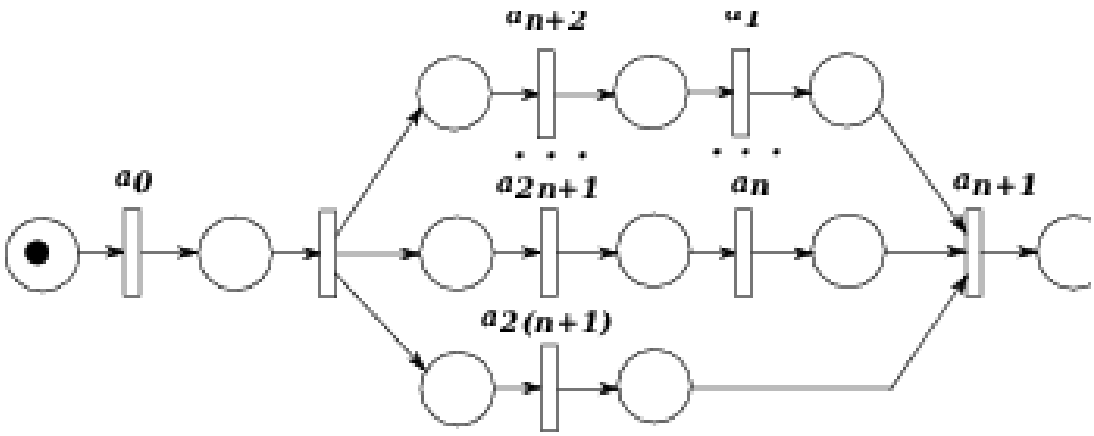

Рис. 2.5. Мережа Петрі, що описує потік задач обробки даних

Особливістю мереж Петрі як апарату моделювання динамічних систем є можливість опису елементів систем на різних рівнях абстракції: від крупно-блокового опису, де переходи моделюють виконання базових операцій, до рівня моделювання черг СМО чи мережевих протоколів взаємодії. При цьому засоби моделювання іваріантні відносно обраної складності опису. Задача «перекладу» різноманітних описів потоків задач на мову мереж Петрі є активно досліджуваною і нині.

Для дослідження потоків задач довільної конфігурації та систем, що ними керують, є доцільним виокремити базові шаблони, що зустрічаються у конфігураціях потоків задач. На рис. 2.6 наведено перелік найбільш поширених базових шаблонів та формальний опис їх поведінки за допомогою мереж Петрі:

а) послідовність (англ. sequence);

б) вічний цикл (англ. loop);

в) розгалуження (розпаралелювання, англ. split, схема «І»);

г) умовний перехід (вибір, англ. choice, схема «АБО»);

д) синхронізація (англ. join, схема «І»);

е) злиття гілок (англ. merge, схема «АБО»).

Комбінація цих шаблонів дає змогу створити більш складні елементи, такі як циклічне виконання з перевіркою умови продовження виконання

циклу (рис.2.7). Якщо потік задач допускає розбиття на складові примітиви, це дозволяє виконати розрахунок часу завершення $t(w) = t(a_N)$, де a_N --- операція, що завершиться останньою або часу виконання потоку $T(w) = t(w) - t_0$, де t_0 --- початковий момент часу), не застосовуючи процедури перекладу всього потоку на мережу Петрі та її подальшого моделювання, а лише знаючи тривалість $T(a_i)$ та правила розрахунку часу завершення $t(a_i)$ складових операцій a_i $(i = 0..n)$ для примітивів, які можна легко вивести.

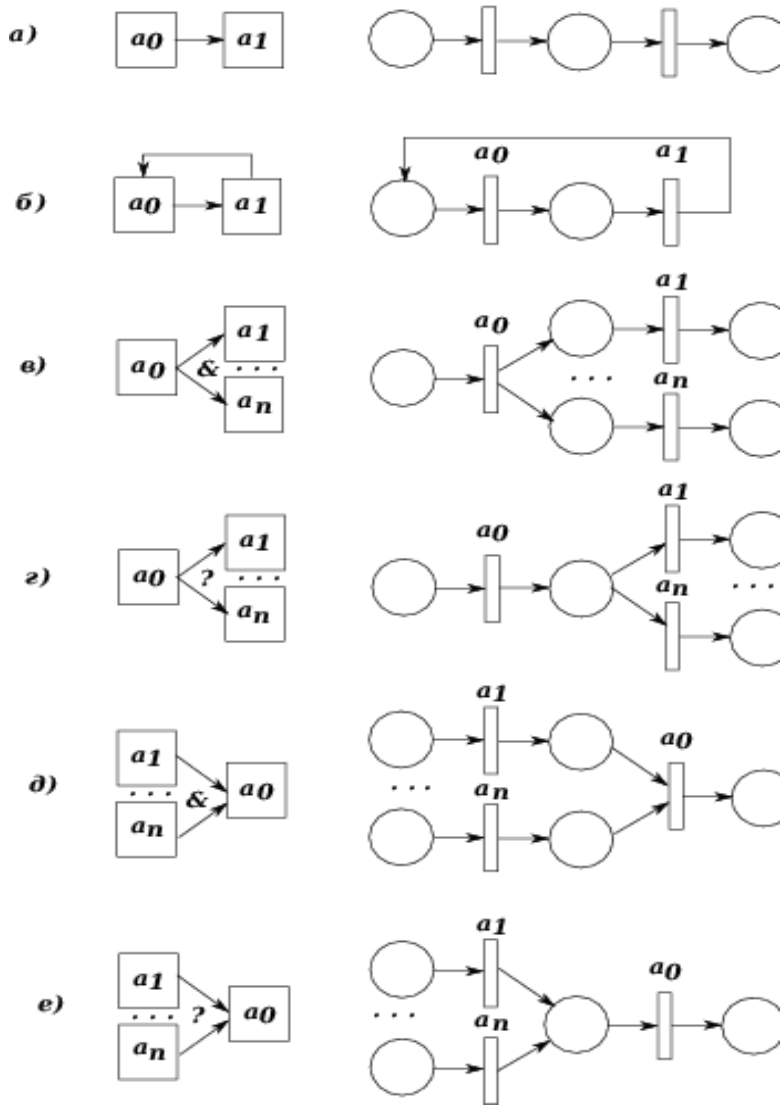

Рис. 2.6. Базові примітиви потоків задач

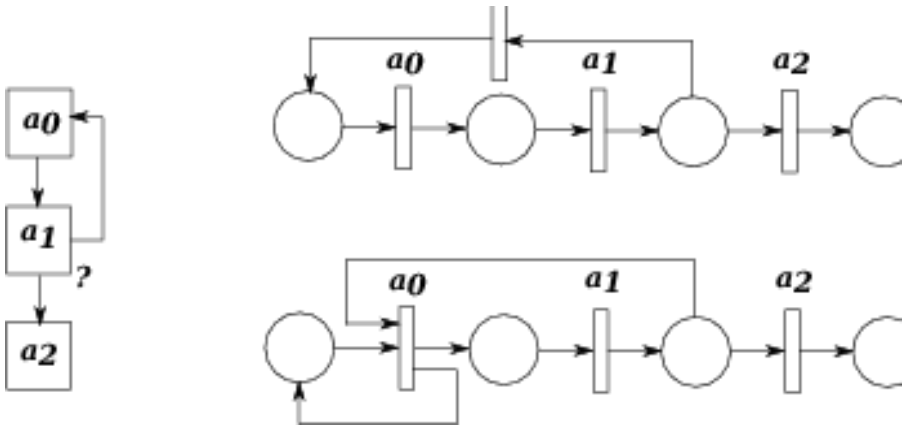

Рис. 2.7. Варіанти опису циклу з виходом за умовою

Для послідовності маємо:

$$T(w_{seq}(a_0, ..., a_n)) = \sum_{i=0}^{N} T(a_i)$$

Для розгалуження маємо:

$$t(w_{sp}(a_0, a_1, ..., a_n)) = \max t(a_i), i = 1..n,$$
$$t(a_i) = t(a_0) + T(a_i)$$
$$T(w_{sp}(a_0, a_1, ..., a_n)) = T(a_0) + \max T(a_i)$$

Для вибору (якщо буде виконуватись операція a_N) маємо:

$$t(w_{ch}(a_0, a_1, ..., a_n)) = t(a_N), 1 \leq N \leq n,$$
$$t(a_N) = t(a_0) + T(a_N)$$
$$T(w_{ch}(a_0, a_1, ..., a_n)) = T(a_0) + T(a_N)$$

Для синхронізації маємо:

$$t(w_{join}(a_1, ..., a_n, a_0)) = t(a_0),$$
$$t(a_0) = \max t(a_i) + T(a_0), i = 1..n$$
$$T(w_{join}(a_1, ..., a_n, a_0)) = \max T(a_i) + T(a_0)$$

Для злиття (якщо виконувалась операція a_N) маємо:

$$t(w_{mrg}(a_1, ..., a_n, a_0)) = t(a_0)$$
$$t(a_0) = t(a_N) + T(a_0), 1 \leq N \leq n$$
$$T(w_{mrg}(a_1, ..., a_n, a_0)) = T(a_N) + T(a_0)$$

Для циклу на *N* ітерацій (рис. 1.13) маємо:

$$t(w_{loop}(a_0, a_1, a_2)) = t(a_2)$$
$$T(w_{loop}(a_0, a_1, a_2)) = N(T(a_0) + T(a_1)) + T(a_2)$$

Окрім безпосередньої оцінки тривалості виконання потоку задач, важливо оцінювати частку часу, що витрачається безпосередньо *на корисні обчислення* $T^+(w)$ відносно загальних витрат часу $T^o(w)$.

$$K_T^+(w) = T^+(w)/T^o(w)$$

Протилежна величина характеризує *частку накладних витрат* на пересилання даних, очікування в чергах та інші побічні дії або простоювання в процесі виконання потоку задач:

$$K_T^-(w) = 1 - K_t^+(w)$$

Так, для розглянутого прикладу потоку задач з обробки даних у сховищі (рис.2.2), для спрощення умовно вважаючи, що $K_T^-(a_i), i = 0..n + 1$

(накладні витрати припадають лише на пересилку даних) часові оцінки будуть наступними:

$$T(w) = T(a_0) + \max_{i=1}^{n}(T(a_{n+1+i}) + T(a_i), T(a_{2(n+1)})) + T(a_{n+1})$$

$$T^o(w) = \sum_{i=0}^{2(n+1)} T(a_i)$$

$$T^+(w) = \sum_{i=0}^{n+1} T(a_i)$$

$$K_T^+(w) = \frac{\sum_{i=0}^{n+1} T(a_i)}{\sum_{i=0}^{2(n+1)} T(a_i)}$$

Тоді, якщо для прикладу прийняти $T(a_i) = \tau$, $i = 0..(n + 1)$, а $T(a_j) = \tau/k, j = (n + 2)..2(n + 1)$, то:

$$K_T^+(w) = \frac{(n+2)\tau}{(n+2)\tau + (n+1)\tau/k} = 1 - \frac{n+1}{(n+2)(k+1)-1}$$

Тоді $K_T^-(w) = \frac{n+1}{(n+2)(k+1)-1}$, тобто частка часу, що витрачається на накладні витрати, в даному прикладі прямо пропорційна відношенню часу обробки даних до часу пересилок даних.

Час виконання операції потоку задач

Оцінка тривалості виконання операції потоку задач *T(a)* є складовою оцінки часу виконання всього потоку *T(w)*, не зважаючи на метод отримання останньої. Передбачення часу виконання операції є

нетривіальною процедурою, що має враховувати множину факторів, часто стохастичної природи, або таких, що важко піддаються формалізації.

$$T(a) = T_{D_I} + \sum_i T_{Q_i}(a) + \sum_j T_{I_j}(a) + \tau(a) + \delta(a) + \sum_k T_{O_k}(a) + T_{D_O}$$

де T_{DI},T_{DO} --- час на передачу вхідних даних до операції та вихідних від неї відповідно: $T_{DI} = d_I / s_I$,$T_{DO} = d_O / s_O$, де d_I, d_O представляє загальний об'єм даних, що передаються до операції та від неї відповідно, а s_I, s_O визначає швидкість каналу передачі даних (пропускна здатність мережі) для завантаження та вивантаження даних відповідно;

T_{Qi} --- час очікування заявки на обчислення в і-й черзі СМО (наприклад, для грід-задач це черги планувальника та ЛСКР --- локальної системи керування ресурсами кластеру). Випадкова величина, що загалом залежить від поточної завантаженості цільових ресурсів системи;

T_{Ij}, T_{Ok} --- час на розбір вхдіних даних та підготовку вихідних даних протоколу j-го або k-го рівня (наприклад, для веб-сервісів це HTTP(S) та SOAP(XML)). Визначається загальним об'ємом даних, об'ємом корисних даних, структурою формату даних у протоколі та конкретним способом їх представленням (кодування, шифрування), продуктивністю ресурсу;

τ -- чистий час на корисні обчислення. Залежить від параметрів алгоритму програми (таких як часова складність), об'ємів оброблюваних даних та продуктивності ресурсу;

δ --- затримки, пов'язані з виконанням забезпечуючих процедур (завантаження програмного модулю, ініціалізація, встановлення мережевого з'єднання, взаємодія з файловою системою та ін. ресурсами ОС, ПЗПР і т.п.).

Згідно прийнятих раніше позначень, $K_{T+}(a) = \tau(a) / T(a)$. У випадку застосування описаного підходу «псевдо-операцій передачі даних», можна вважати, що для таких операцій $T(a_D) = T_{DI}$ або $T(a_D) = T_{DO}$, а для решти обчислювальних операцій $T_{DI} = T_{DO} = 0$.

Аналіз інших властивостей потоку задач

Аналіз конкретних рішень, пов'язаних з інтеграцією СКП в архітектуру КСП, має базуватися на чітко визначених критеріях. Можна навести багато метрик, застосовних для порівняльного аналізу як розподіленого ПЗ (внутрішня та зовнішня зв'язність), так і потоків задач та СКП (цикломатична складність, рівень декомпозиції на операції, їх здатність до компонування та ін.).

Навіть у межах однієї СКП реалізовувати поставлену задачу можна потоками задач різної конфігурації. При цьому важливо як не допустити надмірного ускладнення потоку та наближення складності його складання до складності процесу програмування, так і надмірного спрощення, що суттєво обмежуватиме кількість допустимих конфігурацій потоку та зводитиме нанівець усі переваги від можливості редагування потоків задач.

Кількість операцій в потоці задач. Це, мабуть, найпростіша оцінка складності потоку задач. Її використання може бути виправданим при порівнянні конфігурацій потоку, що досягають однакової мети. Більша кількість операції N не лише ускладнює роботу користувача, але, як правило, означає більшу кількість переходів M між операціями, на які витрачається час.

Цикломатична складність потоку керування. Дана метрика використовується для оцінки складності програмного коду, однак внаслідок того, що вона оперує представленням коду у вигляді графу потоку керування, то є застосовною і до оцінки потоку задач. Вона показує число лінійно незалежних маршрутів у потоці задач і визначається як:

$$Cc(w) = M - N + 2P$$

де M --- кількість ребер (переходів) у графі потоку керування, N --- кількість вузлів у графі (операцій потоку задач), P --- кількість компонентів зв'язності графа.

Для прикладу на рис. 1.9 маємо $M = 3n + 2$, $N = 2n + 3$, $P = 1$, тобто $Cc(w)=n + 1$, що відповідає наявності у потоці задач $n+1$ паралельної гілки. Неважко побачити, що присутність у потоці примітивів-розгалужень призводить до зростання метрики. Очевидно, з кількох рівних за можливостями варіантів реалізації системи керування потоками робіт та її операцій доцільно обирати такі, що ведуть до потоків з меншою цикломатичною складністю, що означає простішу роботу користувачів та меншу кількість їх помилок.

2.3. Керування потоками робіт

Організація сценаріїв прикладний обчислень у вигляді потоків робіт вимагає наявності компонента, який би керував (координував) автоматичним виконанням потоків робіт довільних конфігурацій --- системи керування потоками робіт (СКП).

2.3.1. Життєвий цикл потоків робіт

«Обчислювальний» робочий потік в процесі свого існування проходить через кілька стадій (рис.2.8). Робота починається із формулювання гіпотези або загальної ідеї експерименту, після чого розробник переходить до фази проектування. Ефективність проектування робочих потоків напряму залежить від можливостей із повторного використання готових напрацювань: повторне використання робочих потоків, їх фрагментів та компонентів, де це можливо; використання та наповнення бібліотек готових шаблонів, компонентів та даних. Проектувальник визначає логічну структуру експерименту, обирає програмні реалізації його кроків, зв’язує їх переходами, вирішує питання

передачі даних. На всіх фазах проектувальник взаємодії із засобами системи управління робочими потоками.

Подальша автоматизація кроку проектування можлива за рахунок залучення семантичних технологій, що дозволить автоматизувати пошук компонентів, узгодження форматів даних, композицію компонентів для досягнення поставленої мети.

Фаза проектування поступово переходить у фазу *підготовки потоку*, на якій розробник потоку готує вхідні параметри та завантажує дані для числового експерименту, обирає (якщо є така можливість) ресурси (апаратні, програмні, джерела даних тощо) або вимоги до них (ЦП, пам’ять, ОС, паралельне виконання, виконання у грід і т.д.). Перед виконанням, підготовлений опис потоку бажано проаналізувати (автоматизованими засобами) на наявність вад: синтаксичних, структурних (неініціалізовані змінні, «тупикові» переходи), логічних («вічні цикли», незадіяні відгалуження), та оцінити очікуваний час виконання та імовірність збоїв (надійність).

На *етапі виконання* потоку складений опис потоку передається до менеджера потоку, який відповідає за узгоджене виконання кроків згідно опису, передачу даних від постачальників до споживачів, моніторинг виконання та постачання користувача актуальною інформацією про хід виконання, ведення журналу та корисної статистики, повернення результатів роботи та статистичних даних користувачу.

Після отримання результатів, дослідник має оцінити їх адекватність (із залученням процедур візуалізації результатів розрахунків), та прийняти рішення про необхідність повторення експерименту, після чого життєвий цикл потоку замикається. Аналіз результатів зазвичай відбувається по наступним напрямкам:

- *адекватність*: відповідність отриманих результатів експерименту очікуваним в рамках вхідної моделі;

- *надійність*: помилки, збої, виключні ситуації на різних кроках потоку, їх причини, усунення «вузьких місць» (англ. bottlenecks) в потоці;
- *ефективність*: час виконання, шляхи його поліпшення.

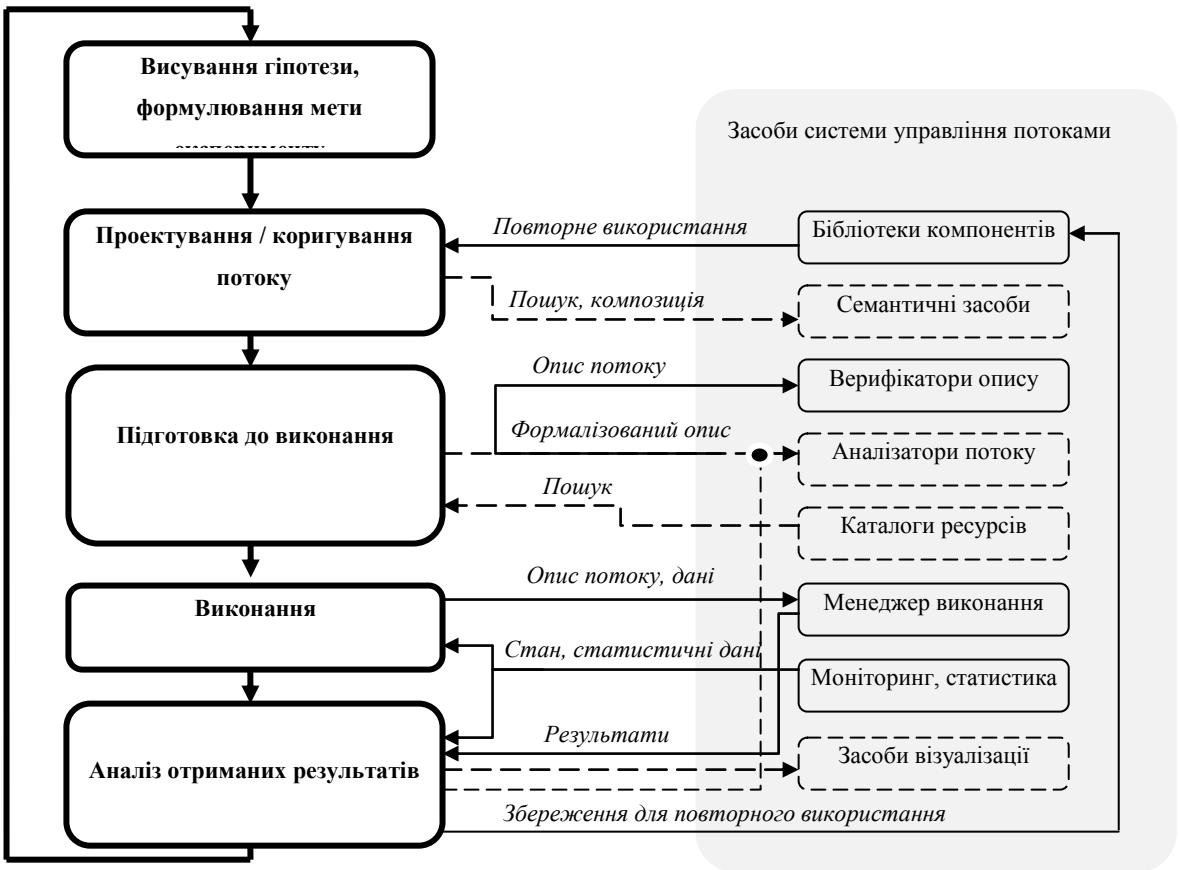

Рис.2.8. Життєвий цикл обчислювального робочого потоку та взаємодія розробника із засобами системи управління потоками на різних фазах циклу (пунктиром показані необов’язкові компоненти)

2.3.2 Програмні засоби керування потоками робіт

Перш ніж переходити до аналізу механізмів узгодження веб-сервісів для виконання потоків робіт, доцільно буде проаналізувати історичний досвід розробок систем керування потоками робіт, включаючи і сервісно-орієнтовані.

Discovery net

Історично першим, пілотним проектом із створення системи управління науковими робочими потоками із залученням високопродуктивних (грід-) обчислень, був проект Discovery net (2001-2005), що фінансувався по програмі UK e-Science; виконавець – Лондонський Імперський коледж. Даний, вже закритий проект береться до розгляду через його внесок у справу визначення базової архітектури систем управління науковими робочими потоками.

Проект брав до уваги, в першу чергу, потреби «наук про життя», моніторингу гео-небезпек, моделювання навколишнього середовища, проблеми пошуку відновлювальних джерел енергії. З часом, завдяки вдалим архітектурним рішенням, що не залежали від профілізації системи, коло застосувань розширилося до біоінформатики, хімії, і навіть - фінансів та бізнесу. В першу чергу, це міждисциплінарне програмне рішення мало слугувати як інструмент аналізу даних (в т.ч. інтелектуального - data mining), зібраних із численних пристроїв та ресурсів, об’єднаних мережею Інтернет або Грід.

Архітектура системи – багатоланкова, верхній рівень якої складає клієнтське програмне забезпечення для швидкої розробки потоків шляхом перетаскування («drag-and-drop») потрібних компонентів. Проміжна ланка представлена сервером потоків, відповідальним за виконання робочих потоків, авторизацію, узгодження потоків із залученими гетерогенними ресурсами, які складають останню ланку архітектури.

Самі робочі потоки описуються на мові DPML (Discovery Process Markup Language – мова розмітки дослідницьких процесів), що є XML-мовою опису як потоків даних, так і потоків управління. Кожен виконуваний компонент потоку має 3 типи портів: вхідні, вихідні та параметри, що задаються користувачем. Кожен порт з’єднується з іншими одним або більше ребром графа (що позначає перехід між кроками потоку).

Цілий потік може бути розгорнутий як окремий компонент зі своїми портами, в цьому випадку представляючи собою один готовий до використання сервіс.

Використання метаданих для кожного компонента, що описують типи даних для вхідних та вихідних портів, дозволяє проводити верифікацію потоків на обов'язкове співпадіння типів з'єднаних входів та виходів (жорстка типізація). Базовим типом даних є таблиця реляційної БД, що складається із набору кортежів зі значеннями полів таблиці. Втім, деякі специфічні застосування зумовили вбудовану підтримку (з компонентами імпорту-експорту та засобами візуалізації) також і додаткових типів, таких як генетичні послідовності чи набори зображень.

Шар потоку управління явно відділяється від потоку даних: окремі фрагменти потоків даних вбудовуються як складові блоки потоку управління. Потік управління описується графом, який складається з вузлів, що організовують загальний хід виконання робочого потоку, додаючи можливості застосування ітерацій та розгалужень (тобто, у графі управління допускаються цикли). Взаємодія між цими вузлами відбувається через обмін командами, а не даними, і виконання потоку управління слідує прямому push-підходу: кожен управляючий вузол визначає, на які зі своїх виходів подати управляючі команди для активізації наступних вузлів. Таким чином, потік управління координує виконання під-потоків даних, причому комунікації відсутні як між шарами потоку управління та потоку даних, так і між окремими потоками даних, поєднаних потоком управління.

Для опису потоків даних вже використовується ациклічний граф залежностей та, відповідно, зворотня pull-модель, в рамках якої для кожного кінцевого вузла графу залежностей ітеративно визначаються вузли-постачальники даних, тобто вимога виконання вузла-споживача ініціює попереднє виконання усіх постачальників.

Triana Workflows

Розробка Університету Кардіффа, початково пов'язана із участю університету у проекті з виявлення гравітаційних хвиль GEO600 (з 2002 року), однак потім успішно перетворилася на міждисциплінарний проект. Зараз до бібліотеки входить більше 500 компонентів для обробки сигналів, зображень, звуку та статистичного аналізу. Серед проектів, що використовували Triana – проекти з дослідження біорізноманіття BiodiversityWorld, медичне моделювання GEMMS (Grid-Enabled Medical Simulation Services), а також проекти з інтелектуального аналізу даних Data-Mining Grid та FAEHIM. Зараз на сайті проекту анонсовано перехід на оновлену, четверту версію.

Компоненти у цій системі іменуються «функціональними одиницями» (units), порти яких з'єднуються між собою через «кабелі» (cables). У якості функціональних одиниць можуть виступати програмні компоненти з різномантіними інтерфейсами: JXTA, P2P sockets, веб-сервіси, грід-сервіси. Ця різноманітність підтримується через проміжні інтерфейсні шари GAP та GAT (Grid Application Toolkit проекту Gridlab). Система написана на мові Java.

Кожна функціональна одиниця може бути відредагована, перекомпільована та заново внесена до бібліотеки компонентів. Також цілий потік може бути згрупований як одна одиниця потоку – для підтримки ієрархічного «вкладання» потоків один в інший. Для реєстрації одиниць потоку використовується XML-опис.

Потік управління базується на «прямому» підході: вихідні дані кожного компоненту передаються «по кабелям» до наступних за порядком виконання компонентів, причому потік управління виконується через спеціальні управляючі повідомлення. На додачу, існують спеціалізовані компоненти для розгалужень та циклів (ForNext, If, Loop, Sleep та ін.).

Потік даних також будується прямим ходом, підтримується як послідовне, так і багатопоточне виконання (компоненти Sequence, Block, Split, Merge). Завдяки їм можливе паралельне виконання кількох функціональних одиниць потоку над різними наборами даних. Присутня опція зі збереження історії обробки даних (History tracking).

Kepler

Наступні дві системи найбільш активно розвиваються і зараз. Перша з них – Kepler (рис.2.9), розробку якої ініціювали представники одразу кількох наукових установ США (Каліфорнійські університети Девіса, Санта-Барбара та Сан Дієго, 2002 рік). Проект є наступником системи моделювання від університету Берклі - Ptolemy II. Зараз коло дисциплін, які підтримує система, значно розширилося: екологія, молекулярна біологія, генетика, фізика, хімія, океанографія, комп'ютерні науки та ін. Серед проектів, що використовують Kepler --- філогенетичні дослідження pPOD, аналіз даних від наукового обладнання у реальному часі REAP (Real-time Environment for Analytical Processing) та інші застосування з аналізу даних та моделювання.

Особливістю архітектури Kepler є те, що механізм виконання потоку відокремлено від конкретної моделі обчислень, а точніше – для кожного робочого потоку встановлюється своя модель обчислень із відповідним менеджером управління.

Примітка. *Термінами Kepler менеджер --– director, тобто «директор» або «режисер». Порівняйте з оркеструвальником бізнес-процесів, про який йтиметься дещо далі).*

Поняття «режисера» є ключовим для системи: якщо актори та їх зв'язки складають модель потоку, то режисери визначають модель

обчислень (наприклад, обробка потоку даних або ж використання поняття часу для задач моделювання). В цьому полягає відмінність, наприклад, від попередньої системи Triana – тут актори «обізнані» лише із типами вхідних та вихідних даних та операціями, що слід виконати над ними, а ось рішення, коли саме викликати того чи іншого актора приймає «режисер».

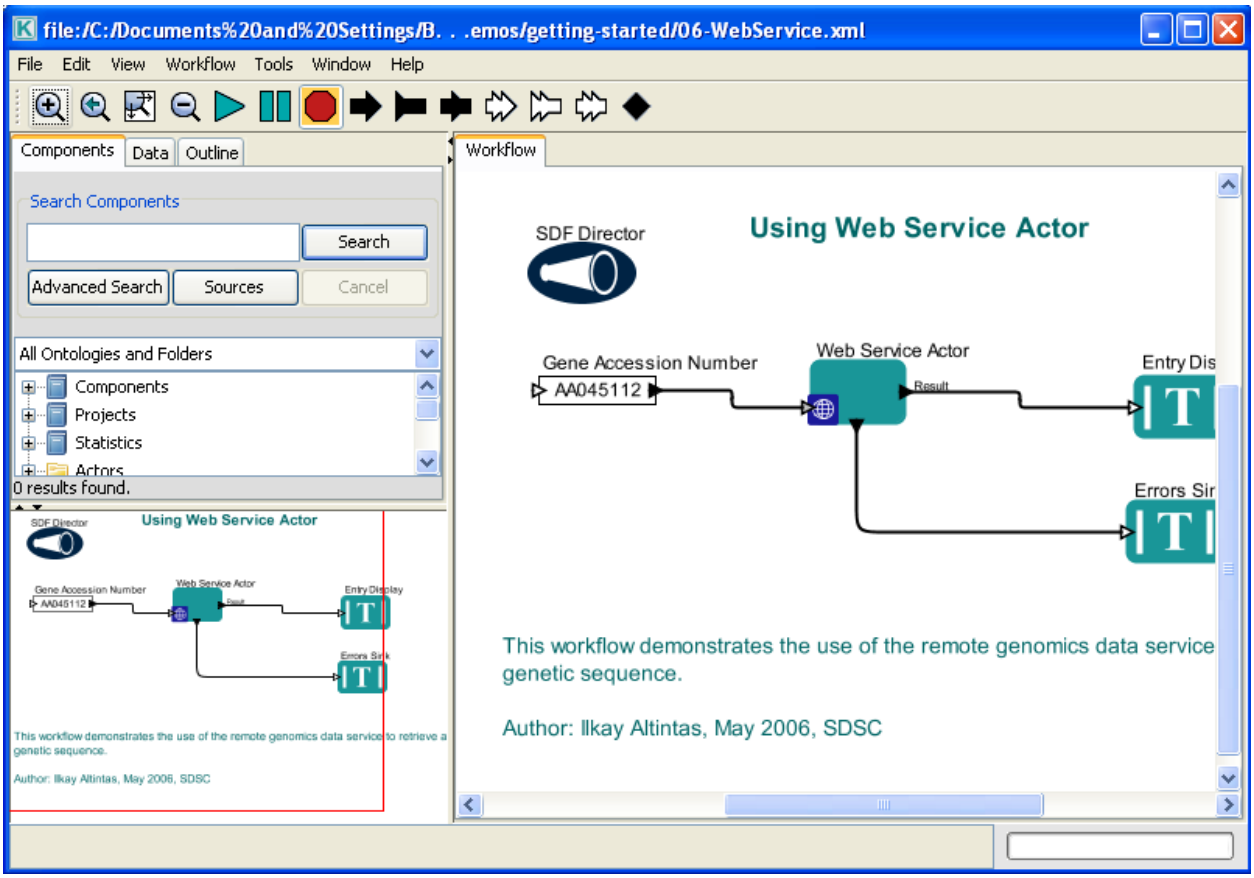

Рис. 2.9. Система управління робочими потоками Kepler: демонстраційний робочий потік із залученням веб-сервісу.

Більш того, для забезпечення ще більшої вигоди від повторного використання, актори також є поліморфними відносно типів даних, тобто операції можуть застосовуватись до різних типів вхідних даних. Особливістю реалізації концепції «актора» також є те, що він може споживати/постачати через свої порти (вхідні, вихідні, змішані) більше одного пакету даних за один виклик. Наприклад, актор, що перетворює

послідовність у масив (sequence-to-array), споживає кілька послідовних наборів даних та продукує з них єдиний масив на виході.

Залежно від сумісності їх режисерів, допускається вкладати потоки один в інший. Мова опису потоків MoML базується на XML. Використовується семантичне анотування потоків та їх компонентів за допомогою онтологій. Система написана на мові Java.

Taverna Workbench

Ця система є одним з головних елементів проекту myGrid (з 2001 р.), виконавцями якого є цілий ряд британських наукових закладів. За розвиток системи управління робочими потоками Taverna відповідає група з університету Манчестера. Первинною метою системи було задовольнити потреби біоінформатиків, які могли будувати робочі потоки з численних віддалених веб-сервісів. Нині проект перейшов під крило Apache.

Отже, ***дана система більше, ніж попередні, орієнтована на використання веб-стандартів, обравши саме WS-інтерфейс як стандартний відкритий інтерфейс компонентів, а не нав'язуючи власний внутрішній опис***. Цим Taverna певною мірою наближається до систем управління бізнес-процесами. Головним компонентом робочих потоків є саме веб-сервіси: SOAP/WSDL або REST, або більш спеціалізовані сервіси BioMart, BioMoby, SoapLab та ін. Поряд із можливістю імпорту та використання віддалених веб-сервісів, бібліотека компонентів також містить і локальні Java-компоненти (система також написана на Java) для виконання деяких технічних задач.

Архітектура системи зазнала деяких змін при переходів від версії 1.7 до 2.0. Втім, загальна концепція лишилася незмінною: графічний редактор Taverna Workbench для створення описів потоків веб-сервісів; менеджер, відповідальний за виконання потоків, та віддалені веб-сервіси як актори - складові потоку. До версії 2.0 використовувалась мова опису потоків

SCUFL та менеджер freeFluo, однак вони згодом були замінені на власний менеджер та XML-мову t2flow. Втім, для користувача така зміна не є суттєвою, оскільки графічний інтерфейс приховує деталі конкретної мови опису робочих потоків, а старі формати підтримуються і у новій версії. Складові сервіси до версії 2.0 іменувалися «процесорами».

Серед архітектурних особливостей також слід відзначити можливість віддаленого контролю за виконанням потоків через сервер Taverna Server. Середовище розробки дозволяє контролювати синтаксичну коректність описів потоків, повторно використовувати потоки, ієрархічно включати потоки в інші потоки. Середовище також пропонує чимало засобів для діагностики, відлагодження та контролю за виконанням, серед яких: моніторинг, збереження історії виконання, призупинення виконання потоку, повторення запитів при відмовах окремих сервісів та ін. Що важливо, підтримуються стандарти безпеки веб-сервісів (робота з x.509-сертифікатами та ін.).

В рамках прийнятої моделі робочого потоку (що, по суті, представляє собою спрямований ациклічний граф – DAG, рис.2.10), кожен потік має свої входи та виходи. Ці порти потоку, а також входи та виходи сервісів-акторів поєднуються зв’язками, які можуть бути двох типів: для передачі даних та координаційні. Останні не призначені для передачі даних між акторами, а лише встановлюють додаткові умови для потоку управління. Виконання потоку відбувається за прямим push-підходом --- від постачальників даних до споживачів. При цьому є можливість виконати потік частково, що зручно, коли виконання деяких сервісів завершилося невдало. Координаційні зв’язки та умовні конструкції є єдиними структурами потоку управління. Цикли підтримуються неявно через потік даних, як буде показано нижче.

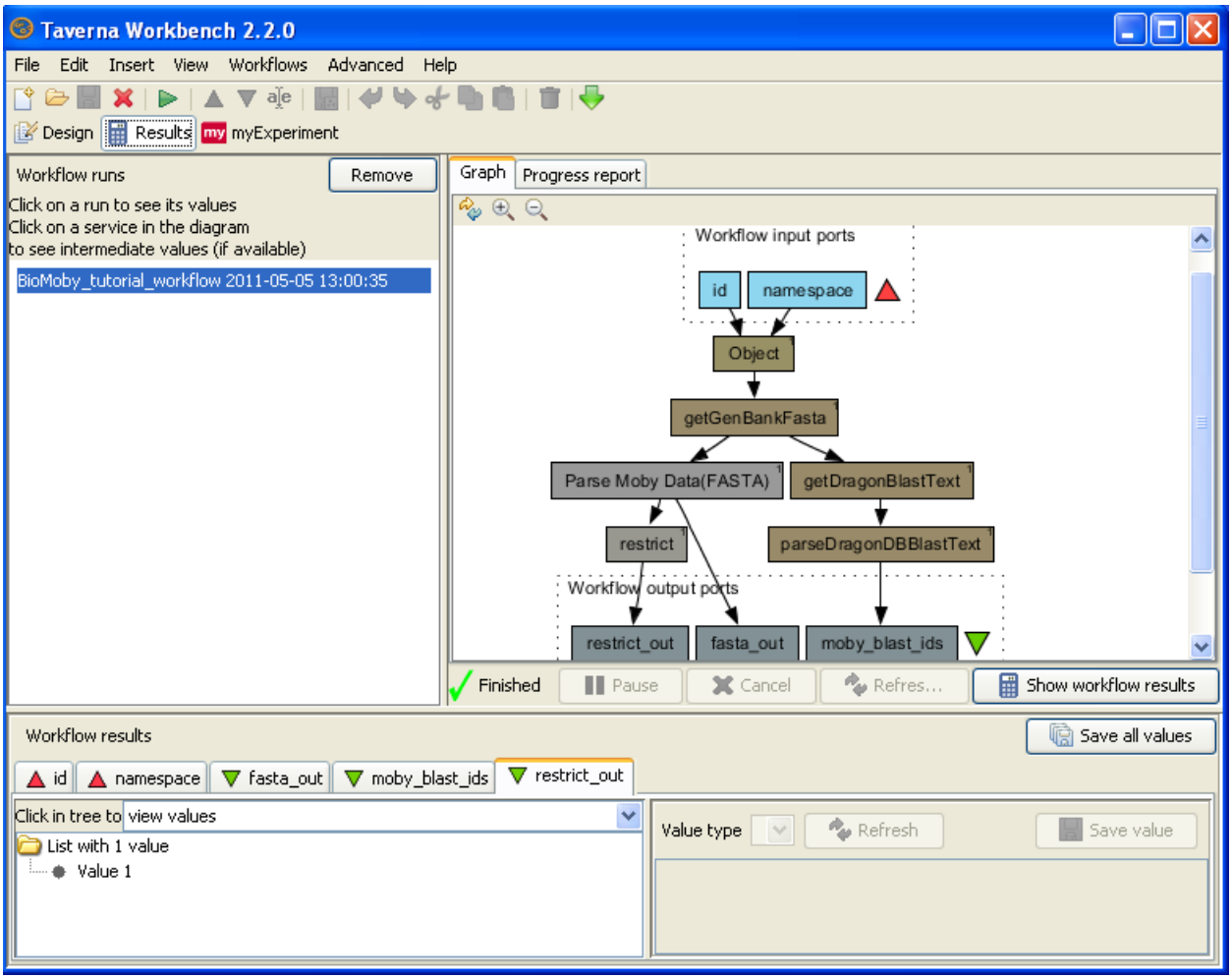

Рис. 2.10. Робоче середовище Taverna Workbench 2.2, режим перегляду прогресу та результатів виконання потоку

Особливістю потоку даних Taverna є специфіка використання тих самих сервісів-акторів як для окремих екземплярів вхідних даних, так і для їх наборів. Нехай сервіс виконує деяку функцію f над вхідними даними x, тоді на виході матимемо f(x). Якщо ж на етапі проектування потоку визначається, що на вхід буде подано масив (точніше, список - list) даних, тобто редактор «знає», що x = [x1, …, xn], то результатом буде вже [f(x1),…,f(xn)] (неявний цикл). Якщо ж на вхід функції подаються два масиви [x1,…,xn] та [y1,…,yn], тоді результатом буде або скалярний добуток виду [f(x1,y1), f(x2,y2),…,f(xn,yn)], або векторний добуток [f(x1,y1),…,f(x1,yn),…,f(xn,y1),…,f(xn,yn)]. Таким чином, агенти потоку не потребують окремих операцій для скалярів та множин.

2.4. Потік робіт як сценарій взаємодії веб-сервісів

На противагу розглянутим системам, що базуються на власних архітектурних рішеннях, мовах опису, графічних редакторах тощо може бути запропонований підхід, що максимально використовує існуючий інструментарій та веб-стандарти. Його суть полягає у використання стандартної мови опису бізнес-процесів WS-BPEL 2.0 для опису робочого потоку, стандартів веб-сервісів у якості специфікацій інтерфейсів компонентів, веб- та грід- сервісів як складових акторів, а також веб-інтерфейсу для віддаленого доступу користувачів до функціональності системи.

Переваги від орієнтації на веб-стандарти полягають у *вищому рівні сумісності системи із іншими стандартними рішеннями*, можливість вибору відкритих стандартних засобів для розробки та реалізації системи: парсери опису, середовище виконання сервісів, менеджери виконання потоків тощо.

Веб-сервіс як інтерфейс компонентів потоку є універсальнішим за компоненти, описані внутрішніми мовами опису Triana чи Kepler, які призначені виключно для використання в одній конкретній системі.

Наслідком такої концепції також є відмова від локального середовища рішення задач на користь веб-інтерфейсу до серверної функціональності. Це дозволяє мати доступ до робочого простору з будь-якої робочої станції, підключеної до мережі Інтернет та оснащеної веб-браузером, а також дозволяє переривати сеанс роботи, не перериваючи процесу виконання потоку.

2.4.1. Хореографія сервісів

Існує два принципово різних підходи до організації узгодженої взаємодії (компонування) веб-сервісів: децентралізований, тобто без координатора (диригента, оркеструвальника) --- т.зв. «хореографія сервісів», та централізований координований підхід --- «оркестрування сервісів».

Перший підхід означає, що веб-сервіси самостійно шукають інші сервіси для взаємодії (приклад – сервіси типу S_C або S_A з «екосистеми сервісів» у першому розділі) та контролюють цю взаємодію. В цьому випадку сервіси мають бути обізнані про склад «екосистеми», що не завжди є доцільним (посилюється зв'язність системи). Крім того, для «хореографії» виявилося важче забезпечити стандартні засоби, аналогічні до наявних та загальноприйнятих стандартів та засобів для «оркестрування сервісів». Хоча слід відзначити спроби стандартизації такого підходу, такі як мова WS-CDL (Web Service Choreography Description Language --- аналог WS-BPEL).

2.4.2. Оркестрування сервісів

Альтернативний підхід --- оркестрування сервісів («калька» із загальноприйнятого визначення англійською --- service orchestration). Особливістю є наявність диригентів-координаторів процесу виконання потоку робіт (сервіси типу S_O з «екосистеми сервісів» у першому розділі). Даний підхід характеризується такою перевагою: сервіси екосистеми «не знають» про існування один одного, тому система відрізняється низькою зв'язністю та простотою підтримки та розширення.

Серед існуючих реалізацій такого підходу найбільше визнання здобули ***бізнес-процеси*** --- будь-яка діяльність, що має вхідний продукт, додає вартість до нього, та забезпечує вихідний продукт для внутрішнього або зовнішнього споживача. В якості продукту --- дані. В якості доданої

вартості --- оброблені, проаналізовані дані. В якості складових кроків діяльності --- виклики веб-сервісів. Суворо кажучи, поняття бізнес-процесу та засоби роботи з бізнес-процесами не обмежуються лише програмною взаємодією сервісів, а й включають взаємодію між людьми, проте ми *зосередимось саме на машинній взаємодії в рамках бізнес-процесів*.

2.4.3. Композитний сервіс як бізнес-процес

Бізнес-процеси є досить близькими до інженерних та наукових робочих потоків, а тому допустимим рішенням може виявитись використання поширених мов опису бізнес-процесів і для опису, скажімо, маршрутів проектування або сценаріїв прикладних обчислювальних досліджень. Втім, для бізнес-процесів одним з головних факторів є стандартизація як шлях до забезпечення функціональної сумісності компонентів від різних постачальників. Зважаючи на це, загальноприйняті стандартні мови опису бізнес-процесів фокусуються саме на описі процесів різного рівня абстракції, а не на можливості дослідження властивостей потоків за їх описом. Тому таким мовам часто бракує чіткості та визначеності, що робить важчим або навіть унеможливлює аналіз потоків: поведінка описаного таким чином потоку не завжди буде однозначною.

WS-BPEL

Одним із таких описових стандартів є BPEL (Business Process Execution Language - мова виконання бізнес-процесів) – стандарт OASIS на опис бізнес-процесів, складених з веб-сервісів (остання версія – WS-BPEL 2.0). Оскільки він орієнтований на веб-сервіси, то технічним деталям виклику сервісів, обробки помилок, передачі сервісів, роботі з WSDL-описами сервісів приділяється значно більше уваги, ніж питанням визначеності результату описаного процесу, його аналізу. Аналізатори BPEL-опису можуть легко перевіряти лише синтаксичну коректність, але не

логічну. Втім, завдяки своїй поширеності та стандарту, на сьогодні розроблено чимало програмних засобів для виконання WS-BPEL-потоків, тому використання BPEL-підходу можна рекомендувати і для систем управління інженерними робочими потоками, елементами яких є веб-сервіси. В цьому випадку для дослідження потоку його опис слід «перекласти» на формальну мову з більшими можливостями з аналізу.

XML-мова WS-BPEL більшою мірою орієнтована на опис потоку управління, за потік даних «відповідають» змінні та їх використання при передачі повідомлень між веб-сервісами. Серед інших, визначені наступні елементи потоку управління: послідовне виконання (Sequence), умовний перехід (If), цикли за умовою (While, RepeatUntil) та ітераційні (ForEach), паралельне виконання через визначення залежностей (Flow) та ін.

Слід зазначити, що поруч із відносно низькорівневими мовами типу BPEL, окрему групу складають графічні мови опису (графічні нотації) бізнес-процесів. Головним завданням цієї групи мов є по можливості інтуїтивно зрозумілий для розробників графічний опис структури робочих потоків. Нечіткістю та слабкою визначеністю опису системи проектування бізнес-процесів часто нехтують, оскільки зависока складність чітко визначених формальних мов часто вважається надлишковою та такою, яку надто складно осягнути розробникам потоків. Елементи таких мов також часто слугують графічною нотацією для інших мов опису бізнес-процесів, таких як BPEL: поширеним є сценарій, коли ***ескіз потоку будується в редакторі з графічних примітивів таких «високорівневих» мов, після чого він уточнюється, деталізується та перекладається на мову BPEL, зрозумілу засобам автоматичного виконання потоків (оркестрування) веб-сервісів.***

Так, добре відома Activity Diagram (AD), одна з ряду UML-діаграм, що описують поведінку, може бути застосована для опису бізнес-процесів та інших робочих потоків. Основним поняттям є дія (action), множина дій

поєднується стрілками в порядку їх виконання. Також доступні елементи, що дозволяють описувати паралельне виконання, вибір, цикли (через примітиви «вибір», «розгалуження», «злиття» див рис.2.11, 2.12). Діаграма представляє собою граф, близький до зображення мережі Петрі, і остання може використовуватись як формальна модель для UML-AD-опису.

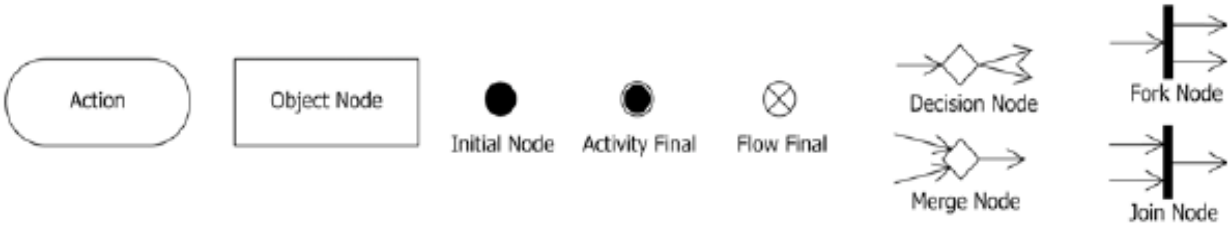

Рис. 2.11. Елементи UML 2.0 AD

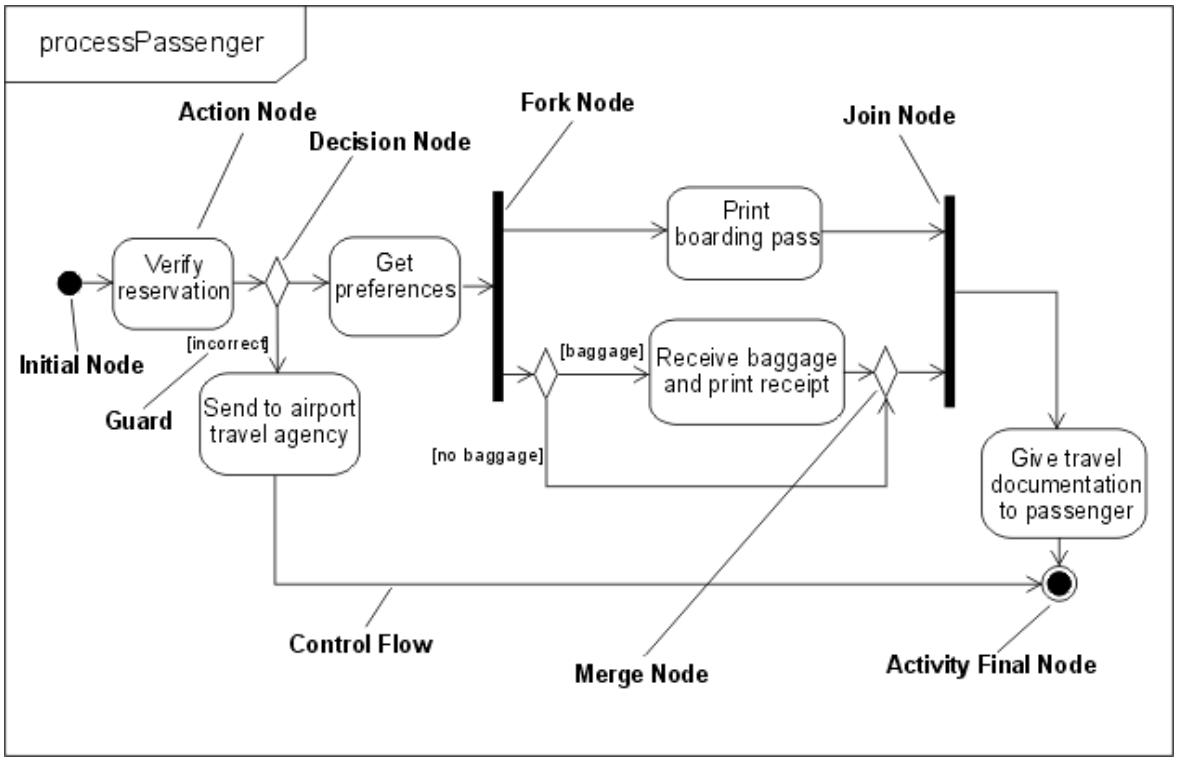

Рис. 2.12. Приклад опису певної діяльності (реєстрації пасажира на авіарейс) на мові UML 2.0 AD

Іншою стандартною нотацією, що призначена для високорівневого опису бізнес-процесів, є BPMN (Business Process Modeling Notation), розроблена Business Process Management Initiative (BPMI). Головною метою

було створити єдину нотацію, легко зрозумілу одночасно як технічним спеціалістам, так і аналітикам та менеджерам. Тобто ця мова відіграє роль проміжної зв'язуючої ланки між проектуванням та реалізацією бізнес-процесу. Набір елементів – подібний до UML-AD (див рис.2.13, 2.14). BPML має допомогти уніфікувати представлення як простих понять бізнес-процесів (відкриті/закриті процеси, навіть --- хореографія процесів), так і більш специфічних (обробка виключень, компенсація транзакцій). Поширеність програмних засобів та добре розроблені шляхи для перетворення BPMN-BPEL робить цю мову принципово придатною і для прикладних систем управління потоками робіт в якості стандартної графічної нотації.

Рис. 2.13. Елементи BPMN-опису

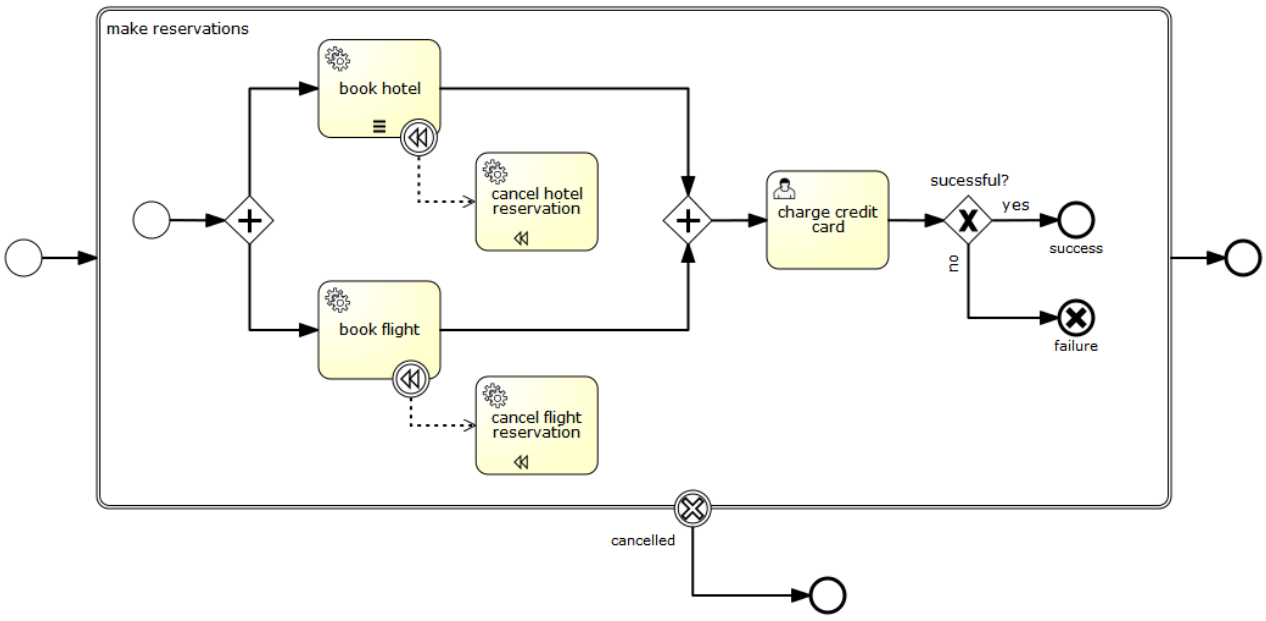

Рис. 2.14. Приклад опису бізнес-процесу нотацією BPMN (резервування авіаквитків та готелю для подорожі)

Намагання суворіше формалізувати опис робочих потоків призвело до появи такої мови, як YAWL (Yet Another Workflow Language, рис.2.15, 2.16), що базується на мережах Петрі. Робочий потік на мові YAWL складається з задач, умов та переходів між ними. Задача може бути простою або включати в себе інші потоки.

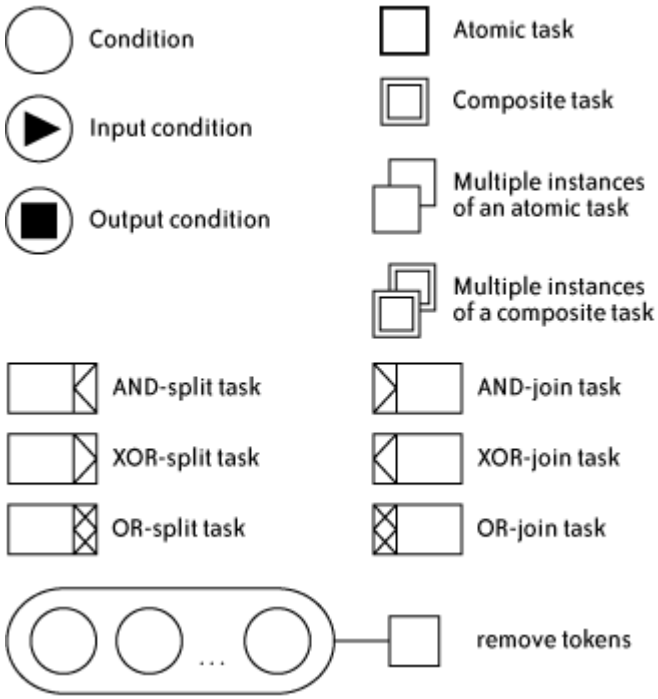

Рис. 2.15. Елементи YAWL-опису

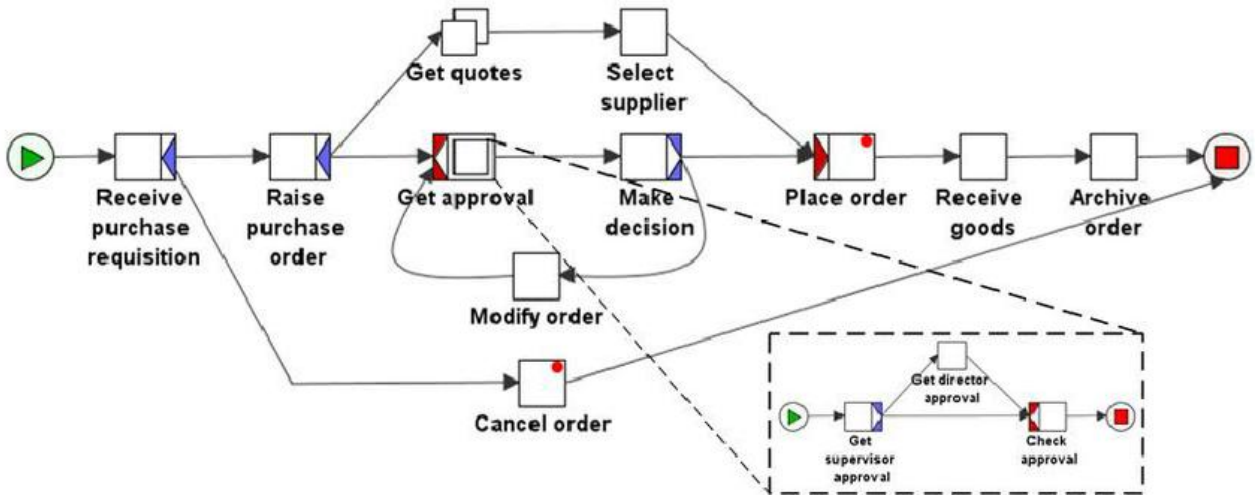

Рис. 2.16. Приклад YAWL-опису деякого процесу (обробка замовлення на купівлю товарів)

Базова модель мереж Петрі була розширена для підтримки кількох сутностей процесів та складних сценаріїв синхронізації. Виокремлено шість явних конструкцій для розгалуження потоку: розгалуження та злиття по схемам ТА, АБО, виключне АБО. При розробці мови досліджено чималий набір характерних шаблонів робочих потоків.

Порівняно складна для опису бізнес-процесів, ця мова втім може поєднувати графічну наочність із потужними можливостями для аналізу робочих потоків. Мова YAWL – основа для однойменної системи управління робочими потоками.

2.5. Розробка композитних сервісів на мові Java

2.5.1. Композитний сервіс фільтрації зображення

Спробуємо реалізувати CustomFilter, що складається з операцій розмиття, інверсії та усереднення, як композитний SOAP-сервіс.

DataURLImageCustomFilterServicePublisher.java:

```
package edu.imageservices.soap.custom;
import javax.xml.ws.Endpoint;
import edu.imageservices.soap.custom.DataURLImageCustomFilterServiceImpl;

public class DataURLImageCustomFilterServicePublisher{
      public static void main(String[] args) {

      Endpoint.publish("http://0.0.0.0:8080/DataURLImageCustomFilterService"
,
                  new DataURLImageCustomFilterServiceImpl());
    }
}
```

Файл логіки DataURLImageCustomFilterServiceImpl.java:

```
package edu.imageservices.soap.custom;
import javax.jws.WebService;
// auto-generated from WSDLs:
import edu.imageservices.soap.blur.DataURLImageBlurFilterService;
import edu.imageservices.soap.blur.DataURLImageBlurFilterServiceImplService;
```

```
import edu.imageservices.soap.invert.DataURLImageInvertFilterService;
import
edu.imageservices.soap.invert.DataURLImageInvertFilterServiceImplService;
import edu.imageservices.soap.merge.DataURLImageMergeFilterService;
import
edu.imageservices.soap.merge.DataURLImageMergeFilterServiceImplService;

@WebService(endpointInterface =
 "edu.imageservices.soap.custom.DataURLImageCustomFilterService")
public class DataURLImageCustomFilterServiceImpl
implements DataURLImageCustomFilterService {
      @Override
      public String applyFilter(String dataURL, String outputFormat) {

            DataURLImageBlurFilterServiceImplService blurServiceImpl =
                  new DataURLImageBlurFilterServiceImplService();
            DataURLImageBlurFilterService blurService =
                  blurServiceImpl.getDataURLImageBlurFilterServiceImplPort();

            DataURLImageInvertFilterServiceImplService invertServiceImpl =
                  new DataURLImageInvertFilterServiceImplService();
            DataURLImageInvertFilterService invertService =
            invertServiceImpl.getDataURLImageInvertFilterServiceImplPort();

            DataURLImageMergeFilterServiceImplService mergeServiceImpl =
                  new DataURLImageMergeFilterServiceImplService();
            DataURLImageMergeFilterService mergeService =
            mergeServiceImpl.getDataURLImageMergeFilterServiceImplPort();

            String dataURLBlurred = dataURL;
            for (int i = 1; i < 10; i++) {
                  dataURLBlurred =
                        blurService.applyFilter(dataURLBlurred, "bmp");
            }
            String dataURLResult = mergeService.applyFilter(
                  dataURLBlurred,
                  invertService.applyFilter(dataURL, "bmp"),
                  outputFormat);
            return dataURLResult;
      }
}
```

Інтерфейс сервісу --- DataURLImageCustomFilterService.java:

```
package edu.imageservices.soap.custom;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.WebParam;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
import javax.jws.soap.SOAPBinding.Use;

@WebService
@SOAPBinding(style = Style.DOCUMENT, use = Use.LITERAL)
public interface DataURLImageCustomFilterService {
      @WebMethod String applyFilter
            (@WebParam(name = "dataURL")      String dataURL,
             @WebParam(name = "outputFormat") String outputFormat);
}
```

Результат виконання – див рис. 2.17

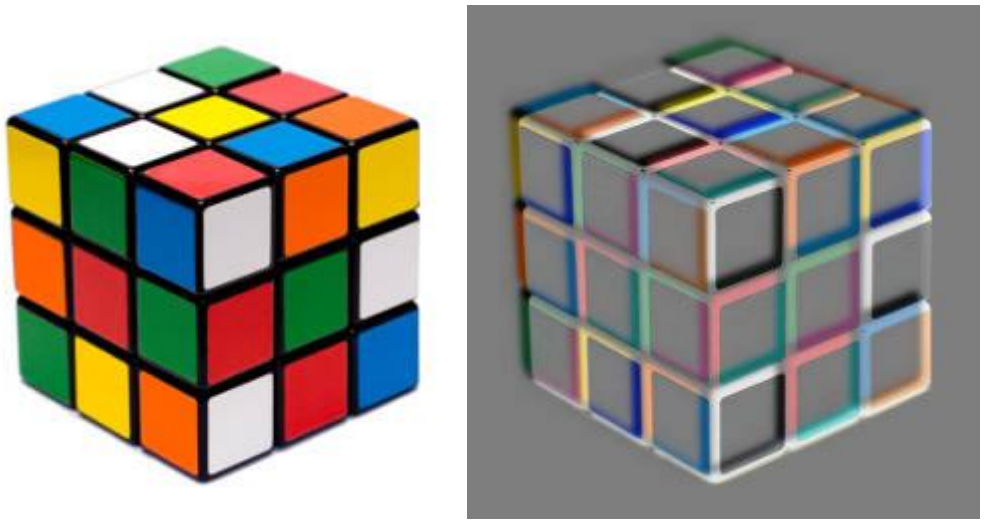

Рис. 2.17 Вхідні дані та результат роботи композитного фільтру

2.5.2. Бізнес-процес фільтрації зображення

Для реалізації фільтру зображення як бізнес-процесу попередньо слід встановити один з доступних BPEL-«двигунів», тобто сервер, який відповідатиме за розгортання бізнес-процесу як веб-сервісу та забезпечуватиме його функціонування.

Зупинимось на Apache ODE, як одному з широковживаних та визнаних інструментів (рис. 2.18.)

Для того, щоб розгорнути бізнес-процес у Apache ODE, слід підготувати набір конфігураційних файлів та, wsdl-описи сервісів, що залучені до процесу, та, звичайно, сам BPEL-опис.

Простий процес фільтрації зображення з двох фільтрів виглядатиме таким чином:

Рис. 2.18. Веб-інтерфейс оркеструвальника Apache Ode

```
<process name="customFilter"
targetNamespace="http://imageservices.edu/bpel/custom"
 xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
 xmlns:img="http://imageservices.edu/bpel/custom">
    <partnerLinks>
    ...partner links description
    </partnerLinks>
    <variables>
        <variable name="blurIn"  messageType="img:string"/>
        <variable name="blurOut" messageType="img:string"/>
        <variable name="invertIn"  messageType="img:string"/>
        <variable name="invertOut" messageType="img:string"/>
    </variables>
    <sequence>
        . . .
        <invoke name="BlurService"
               inputVariable="blurIn"  outputVariable="blurOut"
               partnerLink="BlurService"
               portType="img:BlurServicePort"
               operation="applyFilter"/>
        <assign name="intermediateResult">
          <copy>
             <from>$blurOut.result</from>
             <to variable="invertIn"/>
          </copy>
        </assign>
        <invoke name="InvertService"
               inputVariable="invertIn"  outputVariable="invertOut"
               partnerLink="InvertService"
               portType="img:InvertServicePort"
               operation="applyFilter"/>
        . . .
    </sequence>
 </process>
```

2.5.3. Завдання для самостійної роботи

1. Спробуйте побудувати описи на різних графічних мовах та розрахувати окремі метрики для розглянутого сценарію обробки зображень.

2. Спробуйте реалізувати композитний REST-сервіс аналогічно до SOAP-сервісу з наведеного прикладу. Спробуйте пов’язати його як з SOAP, так і з REST-сервісами.

3. Спробуйте реалізувати набір композитних сервісів для: обробки текстів (переклад, пошук ключових слів, виправлення помилок тощо), математичних операцій (вирішення систем лінійних алгебраїчних рівнянь тощо) або ін. --- на базі п.3. завдань для самостійної роботи першого розділу.

4. Познайомтеся з альтернативними засобами виконання бізнес-процесів, такими як Apache Taverna, OW2 Orchestra, WSO2 ESB, bpel-g та ін., спробуйте реалізувати з їх допомогою сценарій фільтрації зображень.

Висновки по розділу

1. Одною з головних переваг СОА є здатність сервісів до автоматизованого узгодженого виконання --- компонування, що дозволяє надати новий фунціонал, споживаючи вже доступний у вигляді існуючих веб-сервісів, в т.ч. від різних постачальників.

2. Компонування сервісів можливе у двох формах --- некоординованій (хореографія) та координованій (оркестрування). При оркеструванні досягається менша теоретична зв’язність системи, оскільки сервіси не мають бути обізнані про існування інших сервісів --- за них організацією взаємодії займається «оркеструвальник».

3. Найбільш поширеним в промисловості нині є стандарт WS-BPEL 2.0 для організації взаємодії сервісів.

4. Бізнес-процеси можуть створюватись не лише людиною, а й генеруватись автоматично для динамічного компонування сервісів під конкретні вирішувані задачі.

Контрольні питання

1. Що таке потік робіт? Його метрики?
2. Як пов’язаний потік робіт та бізнес-процес?
3. Як пов’язані бізнес-процес та веб-сервіси?
4. Які мови опису сценаріїв взаємодії веб-сервісів Ви знаєте? В чому їх сильні та слабкі сторони?

Розділ 3. СОА і семантичні технології, грід та хмари

Даний розділ знайомить читача з окремими аспектами подальшого розвитку СОА-підходу в сучасних умовах, коли гостро постає питання автоматизації пошуку потрібних сервісів та компонування з них бізнес-процесів, використання СОА для доступу до високопродуктивних ресурсів та хмарних ресурсів.

3.1. Ресєстрація сервісів

Реєстрація веб-сервісів є однією з важливих процедур для гнучких СОА-додатків. Стандартним рішенням для організації пошуку доступних веб-сервісів є UDDI-реєстр.

UDDI (Universal Description Discovery & Integration, універсальний опис, виявлення, інтеграція) є стандартом на механізм для пошуку сервісів. UDDI надає відносно незалежний механізм для публікації та пошуку описів сервісів. У специфікації визначається API (програмний інтерфейс) реєстру, причому інтерфейс призначений виконувати дві важливі групи задач: 1) реєструвати постачальника та його сервіси, та 2) шукати зареєстровані сервіси та зв’язуватись із ними. Тобто, вузол UDDI-реєстру служить як провайдеру сервісу (публікуючи сервіси), так і клієнтам сервісу (надаючи можливість проглядати каталог веб-сервісів, шукаючи певний сервіс та «прив’язуючи» клієнта до нього). І реєстрація, і опитування здійснюються через визначені UDDI-команди, які передаються через протокол SOAP.

Реалізація UDDI насправді представляє собою ще один веб-сервіс, що дозволяє своїм клієнтам реєструвати нові інтерфейси веб-сервісів на вузлі, проглядати їх перелік, перевіряти їх властивості, прив’язуватись до зареєстрованих веб-сервісів. Для доступу до цих сервісів UDDI клієнт надсилає SOAP-повідомлення у термінах UDDI-схеми. SOAP-запит

надсилається до серверу та десеріалізується SOAP-процесором на UDDI-вузлі. Далі виконуються UDDI-запити до реєстру, що перетворюються на запити до бази даних. Відповідь зворотнім порядком через SOAP-процесор для серіалізації та веб-сервер доставляється клієнту.

Відносну популярність дістало створення власних приватних реєстрів для корпоративного інтранету або B2B-мережі. Хоча публічні реєстри різного часу створювались на ibm.com, microsoft.com, sap.com, uddi.org, xmethods.com, згодом компанії зосередились на приватних реєстрах, використовуючи UDDI як стандартний механізм реєстрації та пошуку корпоративних сервісів у внутрішній мережі або для представлення своїх сервісів бізнес-партнерам.

Реєстри UDDI зберігають інформацію про організації, їх сервіси, і про те, як до цих сервісів отримати доступ. Стандартизованими є модель даних та API які дозволяють публікацію такої інформації та її опитування. Така система часто порівнюється із електронною телефонною книжкою, у якій інформація різних типів проіндексована та представлена відповідним чином. UDDI підтримує три типи даних реєстру:

Білі сторінки – функції UDDI як «білих сторінок» дозволяють шукати постачальника по імені або якомусь іншому унікальному ідентифікатору типу DUNS чи Thomas Register. Ця інформація доступна через елемент businessEntity.

Жовті сторінки – категоризація. UDDI підтримує категоризацію по галузях промисловості, продукції, місцезнаходженню. Ця інформація також пов’язана з елементом businessEntity.

Зелені сторінки – зареєстровані записи по типам сервісів та функціоналу, який вони надають. Ці можливості надаються елементами businessService та bindingTemplate.

Тобто, загалом, UDDI дозволяє реєстрацію та пошук за такими критеріями: назва, ідентифікатор, галузь, продукція (послуги),

місцезнаходження, тип сервісу. Розглянемо дещо детальніше структури даних реєстру. Уся інформація в реєстрі зберігається та модифікується за допомогою взаємопов'язаних екземплярів кількох структур даних:

1. *businessEntity* (інформація про організацію-постачальник сервісу)
2. *businessService* (опис функціональності сервісу)
3. *bindingTemplate* (технічні деталі сервісу)
4. *tModel* (атрибути чи метадані про сервіс, такі як таксономії, транспорт, цифрові підписи тощо)
5. *publisherAssertions* (відношення між сутностями в реєстрі)
6. *subscription* (запит на внесення змін до списку сутностей)

Кожна структура даних в реєстрі має унікальний ключ – UUID (Universally Unique ID). Що важливо, реєстр дозволяє побудову різних таксономій для створення семантичної структури інформації, яка зберігається в реєстрі.

Розглянемо основні процедури по роботі з таким реєстром.

Реєстрація сервісу

Інформація, яка використовується при реєстрації, складається з чотирьох типів структур даних (на відміну від другої версії UDDI, де їх стало вже п'ять). На рис.3.1 показано, як ці структури утворюють специфікацію UDDI.

Кожна з цих чотирьох XML-структур містить ряд полів. Докладне пояснення кожної з цих структур можна знайти в довідковому документі щодо структур даних UDDI. Ці структури також описані в специфікації UDDI Programmer's API.

Як видно з рисунка (суцільні жирні і тонкі лінії), структура businessEntity містить одну або декілька унікальних структур businessService. Аналогічно, окремі структури businessService вміщають певні екземпляри даних bindingTemplate, які в свою чергу включають

вказівники на певні екземпляри структур tModel. Слід розглянути кожну з цих структур окремо.

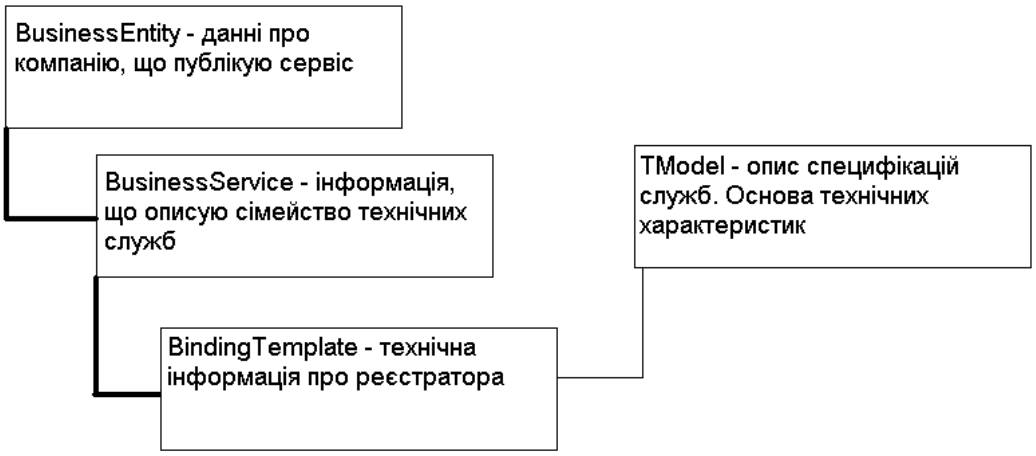

Рис.3.1. Структура специфікацій UDDI

BusinessEntity (бізнес-сутність): дана структура охоплює інформацію про постачальника сервісу і використовується компанією-постачальником для опису та публікації інформації про себе і про пропоновані послуги. З точки зору XML, businessEntity - структура даних верхнього рівня, в якій виражаються опис послуги та технічна інформація. Структура businessEntity має унікальний ключ, який пов’язує її з даними, які зазвичай знаходяться в «Жовтих» і «Білих сторінках».

Приклад:

```
<businessEntity authorizedName="0100003PAJ"
  operator="www.ibm.com/services/uddi"
  businessKey="83B31400-7581-11D5-889B-0004AC49CC1E">
<discoveryURLs>
  <discoveryURL useType="businessEntity">
     http://www-3.ibm.com/services/uddi/
     testregistry/uddiget?businessKey=
     83B31400-7581-11D5-889B-0004AC49CC1E
  </discoveryURL>
</discoveryURLs>
<name>CAD dept. NTUU KPI</name>
<description xml:lang="en">
  NetAllted job submit
</description>
```

```
<contacts>
  <contact useType="CEO,COO">
    <description xml:lang="en">
      Using circuit design to simulate mechanical components
    </description>
    <personName>Ivan Ivanov</personName>
    <phone useType="Work">
      123-456-7890</phone>
    <email useType="">
      admin@cad.kiev.ua</email>
   </contact>
</contacts>
```

BusinessService (інформація про сервіс): ця структура позначає послуги, що надаються businessEntity. Зазвичай, вона містить унікальний ключ, що використовується для представлення цієї послуги, а також зрозуміле людині ім'я служби, необов’язковий опис і структури bindingTemplate, які містять технічну інформацію.

BindingTemplate (зв'язуючий шаблон): дана структура представляє дані, необхідні для опису технічних характеристик реалізації даної служби. IT-фахівці через неї отримують в своє розпорядження інформацію, що вимагається для виклику служби. Кожен bindingTemplate має одного логічного «батька» businessService, який у свою чергу має одного «батька» businessEntity. Крім того, у кожного bindingTemplate є унікальний зв'язуючий ключ, ключ до служби (associate service key), і, найголовніше, - accessPoint і tModelInstanceDetails. Елемент accessPoint представляє адресу для виклику цієї Web-служби. Це може бути URL, адреса електронної пошти (або - ви не повірите - просто телефонний номер).

Основне завдання tModel (інформація про специфікації для постачання служб) - представляти технічну специфікацію. У специфікації UDDI написано: «Мета tModel в межах UDDI-реєстру - забезпечення довідкової системи, заснованої на абстракції». Вона приймає форму «метаданих по ключу (Keyed metadata)»; tModel включає ключ, ім'я, додатковий опис і URL, за яким можна отримати інформацію про дані.

Посилання на tModel - запорука того, що компанія, публікуючи веб-сервіс, реалізувала його так, що він сумісний з цією tModel. Таким чином, більшість компаній може надавати веб-сервіси, сумісні з одними і тими ж специфікаціями.

UDDI підтримує два великих класи API: Publish API та Inquiry API. Publish API пропонує механізм, за допомогою якого постачальники сервісів можуть зареєструвати себе і свої сервіси в UDDI-реєстрі. Інтерфейс Inquiry API дозволяє шукати доступні сервіси і забезпечує два типи викликів: механізм пошуку (алгоритм «поглиблення», за допомогою якого можна почати пошук з самого високого інформаційного рівня і продовжити заглиблюватися, поки не знайдена необхідна інформація) та механізм вилучення, застосовуваний, коли вся інформація, яку необхідно знайти в службі, є в наявності.

Повідомлення в Publish API представляють собою команди, які використовуються для опублікування та оновлення інформації, що міститься в UDDI-сумісному реєстрі. Кожна компанія повинна спочатку вибрати один сайт-оператор (Operator Site), який буде зберігати її інформацію. Після того як він обраний, інформацію можна буде оновлювати. Виклик методів у Publish API вимагає авторизації і виконується зазвичай в рамках механізму захисту. HTTPS використовується для всіх методів у Publish API. Цей інтерфейс включає наступні функції:

1. Чотири повідомлення для збереження кожної з чотирьох структур: save_business, save_service, save_binding, save_tModel. Кожне з цих повідомлень типу «зберегти *» (save_*) приймає в якості вхідних даних authToken і одну або кілька відповідних структур. Наприклад, save_business бере в якості входу одну або кілька структур businessEntity. Або ж кожна така функція "зберегти" буде також приймати в якості параметра елемент uploadRegister, який

розгортається в адресу HTTP URL. URL повинен повертати «чистий» XML-документ, який містить тільки відповідну структуру, очікувану методом «зберегти».

2. Чотири повідомлення для видалення кожної з чотирьох базових структур: delete_business, delete_service, delete_binding, delete_tModel. Всі вони приймають відповідний ключ UUID (унікальний ідентифікатор) як параметр. Успішне завершення або збій видалення вказується у відповідному коді, що повертається в SOAP-структурі dispositionReport. Наприклад, для того, щоб видалити «бізнес», треба передати таку структуру XML:

```
<delete_business generic="1.0" xmlns="urn:uddi-org:api" >
  <authInfo>Some auth info here </authInfo>
  <businessKey>83B31400-7581-11D5-889B-0004AC49CC1E
  </businessKey>
</delete_business>
```

3. get_authToken: ця функція - програмний аналог логіна (login). Вона використовується для запиту токена аутентифікації (англ. authentication token) у UDDI-оператора. Всі функції Publish API вимагають використання цього токена. Слід мати на увазі, що ця функція вважається необов’язковою, і тому UDDI-оператори, що надають інший механізм аутентифікації - наприклад алгоритм, заснований на реєстрації для користувача ID / паролів - необов'язково її реалізують;
4. discard_authToken: дана функція використовується для інформування UDDI-реєстру про те, що authToken більше не є діючим;
5. get_registeredInfo: ця функція призначена для видачі короткого резюме про всю інформації (особливо, про ключі businessEntity і tModel).

Пошук сервісу

Повідомлення даного типу в Inquiry API представляють запити, з якими можна звертатися до UDDI-реєстру, і можуть бути розподілені за двома категоріями, а саме: перегляд і деталізація:

1. Перегляд: UDDI підтримує чотири типи звернень для цієї моделі, як не важко здогадатися - по одному для кожного з чотирьох типів даних. find_business зазвичай, викликається, коли доступна будь-яка приватна інформація, і повертає структуру businessList, яка є скороченим набором ключової інформації зі структури businessEntity. Для відповідного збігу можна кілька разів проходити через businessList, а потім використовувати повідомлення find_service для того, щоб отримати окремі типи служби. Кожне з цих «пошукових» (find_*) звернень приймає також необов'язковий елемент findQualifiers, який змінює метод пошуку, що використовується за замовчанням. Прикладами findQualifiers є «exactNameMatch», «caseSensitiveMatch», «sortByNameAsc» і так далі. Ці пошукові виклики підтримують атрибут maxRows, який визначає максимальне число результатів пошуку, що повертається;
2. Деталізація: якщо доступний ключ для одного з чотирьох основних типів даних, цей ключ можна використовувати для отримання інформації, що стосується окремого примірника цього типу даних. Цей тип виклику – «отримати» (get_*) - застосовується для отримання інформації, що відноситься до будь-якого типу даних для даного ключа. Конкретніше, це виклики: get_businessDetail, get_serviceDetail, get_bindingDetail, get_tModelDetail і, нарешті, get_businessDetailExt, який надає докладну інформацію про businessEntity.

Для того, щоб викликати службу, зареєстровану в UDDI-реєстрі, необхідно, щоб програма-клієнт «знала» про її існування. Ці дані зазвичай включаються в програмний клієнт під час розробки. Подібні дані про виклик, як вже показано, містяться в типі даних «bindingTemplate», їх необхідно кешувати і використовувати під час виклику; таким чином, при повторюваних зверненнях до служби, описаної за допомогою bindingTemplate, необов'язково повторно звертатися до UDDI-реєстру. Цей механізм, однак, може призвести до ситуації, коли віддалена служба виявилася б модифікованою без повідомлення про це клієнта (наприклад, під час заміни ПЗ на нову версію). Щоб коректно вийти з такої ситуації, необхідно:

1. Викликати передачу get_bindingDetail кешованої bindingTemplate;
2. Перевірити дані, повернуті get_bindingDetail, і викликати службу, використовуючи «новий» bindingTemplate, якщо дані, повернуті get_bindingDetail, відрізняються від кешованої інформації;
3. Якщо виклик виявився вдалим, замінити кешовану інформацію на нову.

Інформаційна безпека

У UDDI забезпечення політики безпеки веб-сервісів не описується. Однак безпека в обміні даними з веб-сервісами важлива тому, що більшість транзакцій здійснюються через «небезпечну» зону - Інтернет. Ще одне обмеження пов'язане з тим, що транзакції з веб-сервісами більше зачіпають взаємодії типу «процес-процес», аніж «людина-процес», отже, в даному випадку динамічна схема забезпечення безпеки набуває ще більшого значення. У зв'язку з тим, що використання веб-сервісів постійно розширюється, кількість учасників середовища веб-сервісів буде невпинно зростати, отже, потрібен масштабований і адаптивний механізм забезпечення безпеки. Такі стандарти, як WS-Security (стандарт безпеки веб-

сервісів), XACML (Extensible Access Control Markup Language, розширювана мова розмітки управління доступом) і SAML (Security Assertion Markup Language, мова розмітки умов безпеки) описують реалізацію забезпечення безпеки на рівні сервісу, але не пропонують ефективного способу поширення інформації з безпеки у мінливих сценаріях взаємодії. В цьому може допомогти UDDI.

Щоб забезпечити захищеність веб-сервісу, зазвичай використовуються маркери безпеки, наприклад, маркери імені користувача, маркери XML, виконавчі маркери, маркери Kerberos, сертифікати безпеки, наприклад, сертифікати X.509, цифровий підпис та шифрування та ін. Основна увага приділяється забезпеченню безпеки обміну повідомленнями SOAP між двома сторонами. Такі стандарти, як WS-Security і WS-SecurityPolicy надають механізм реалізації забезпечення захисту веб-сервісів. WS-Security дозволяє доповнити повідомлення SOAP інформацією з питань безпеки. WS-SecurityPolicy - інфраструктура WS-Policy, розроблена провідними компаніями галузі, в тому числі IBM, Microsoft і Verisign, - надає модель загального призначення і відповідний синтаксис для опису та обміну даними про політики Web-сервісу. WS-Policy визначає базовий набір конструкцій, які інші специфікації веб-сервісів можуть використовувати і доповнювати, щоб описувати широкий діапазон вимог, параметрів і функцій сервісу. WS-SecurityPolicy описує спосіб, яким веб-сервіси можуть «висловлювати» свої вимоги і умови, що мають відношення до забезпечення безпеки. Файл WS-SecurityPolicy, асоційований з веб-сервісом, декларує спосіб забезпечення безпеки з точки зору використовуваних типів маркерів, криптографічних алгоритмів і механізмів. Використання механізму забезпечення безпеки на транспортному рівні є частиною загальної конструкції і допускає еволюцію з плином часу. Файл політики безпеки інформує споживача, запитуючого сервіс, про посвідчення безпеки, якими володіє сервіс, щоб споживач міг здійснити доступ відповідним

способом. Механізми забезпечення безпеки можуть бути різними, тому необхідно відокремити їх від фактичного опису «бізнес-сервісу».

Як правило, інтегровані додатки або системи «бізнес-бізнес» мають створені вручну угоди про політику безпеки. Генератор, використовуваний для створення сервісу, надає набір заходів забезпечення безпеки, а потім клієнт слідує цим заходам при доступі до сервісу. У довгостроковій перспективі виконання встановлених угод є непростою справою. Це призводить до:

1. Помилок ручного вводу від необізнаних учасників;
2. Додаткових витрат на повторну реалізацію процесів;
3. Тенденції до переглядів політики безпеки на стороні сервера з метою уточнення.

Коли компанії вносять свої сервіси до реєстру UDDI, в ньому зберігається URL WSDL в структурі tModel і кінцева точка доступу до сервісу в шаблоні зв'язування. Споживач сервісу може виконати пошук за реєстром і звернутися до кінцевої точки та URL WSDL. Точно так само в реєстрі можна передбачити можливість зберігання файлів політики і безпеки сервісу; передбачається, що споживач-запитувач сервісу, вміє адаптуватися до його політики безпеки.

З веб-сервісом асоціюється файл WS-SecurityPolicy, що визначає політику безпеки, якій слідує даний конкретний веб-сервіс. Споживач-запитувач сервісу, повинен дотримуватися цієї політики в своєму запиті при виклику веб-сервісу. У реєстрі UDDI потрібен механізм, який дозволяв би споживачеві знаходити політику безпеки веб-сервісу аналогічно тому, як це робиться при пошуку WSDL. Висувались пропозиції зберігати в реєстрі UDDI URL файлу політики безпеки точно так само, як зберігається URL WSDL в елементі tModel. У tModel URL WSDL зберігається в елементі OverviewURL. Оскільки допускається мати будь-яку кількість

OverviewURL, можна зберегти файл політики безпеки в іншому OverviewURL в тій же структурі tModel.

Файл політики можна вилучати програмно так само, як і WSDL, крім того, можна використовувати його під час виконання на клієнтській машині під час виклику Web-сервісу. Клієнт повинен вміти застосовувати політику безпеки, пропоновану файлом політики безпеки.

Це дає обом сторонам гнучкість у розробці політики безпеки з урахуванням своїх конкретних вимог. Для оперативного обміну інформацією не потрібно мати встановлену угоду на рівні сервісу (service-level agreement, SLA). Цей процес вже вбудований в архітектуру, тому дозволяє обом сторонам зв'язуватись між собою без зайвих проблем. Вони обмінюються деяким набором загальних припущень про підтримку певних політик чи протоколів.

3.2. Семантичне розширення реєстру

Невпинне наростання інформаційних потоків є візитною карткою сьогодення. Проте, ставши на шлях до інформаційного суспільства, людина зацікавлена аж ніяк не у інформації, а в знаннях, що закладені у неї. Дотепер проблема аналізу інформаційних потоків майже не стосувалася пересічних людей, однак і вони зараз гостро відчувають перевантаженість інформацією, чому посприяв стрімкий розвиток Інтернет. Кількість документів, доступних для всіх у Всесвітній мережі, є визначним досягненням, однак ця кількість робить задачу пошуку занадто обтяжливою та витратною у часі. І що засмучує, людині в цьому пошуку майже не здатна допомогти машина. Адже всі накопичені документи створені людьми і для людей, на природній мові, допоки «не зрозумілій» комп’ютерам. Такий сценарій був давно очевидний для і творців Інтернет, які останнім часом активно намагаються

впровадити семантичні технології під привабливими «вивісками» Веб 2.0 та Веб 3.0.

Звичайно, Інтернет-пошук – не єдине застосування семантичних технологій. Так, наприклад , все більшого поширення набувають веб-сервіси. Ця ж технологія була запозичена і для побудови Грід-інфраструктур. Але задачу автоматизації пошуку веб-сервісів, їх поєднання, побудови з них складних потоків неможливо вирішити, не проаналізувавши основні властивості сучасних семантичних технологій та їх застосування для опису прикладних сервісів з репозиторію.

В інформаційних технологіях формалізований опис знань у деякій предметній області здобув назву *онтології*. Формалізований – значить, зведений до мови, яку здатна обробити машина. Це дозволяє перекласти частину роботи з упорядкування, фільтрації, аналізу даних на комп'ютер у тих областях, де він досі відігравав пасивну роль браузера чи друкарської машинки.

Онтологія в інформаційних технологіях – це структурована сукупність понять, атрибутів, відношень, екземплярів, що має на меті формалізацію знань у певній предметній області. Мета формалізації – уможливлення машинного аналізу знань, задля чого онтологія має бути описана певною мовою у певному форматі. Машинний аналіз онтології має на меті автоматизованє надання певних логічних висновків про знання, що моделюються цією онтологією. Це додасть нові, «інтелектуальні» можливості для системи завдяки застосуванню описаних знань та автоматизувати підбір сервісів під вимоги кінцевого користувача.

Що можуть дати семантичні технологія для СОА? Один з варіантів --- автоматизація пошуку сервісів шляхом розширення реєстру. UDDI-реєстри можуть бути розширені анотаціями в рамках онтологій веб-сервісів для автоматичного (без участі людини) пошуку потрібних сервісів під задану задачу чи під наявні формати даних.

Тут доцільно розглянути деякі аспекти семантичних технологій, що можуть бути використані при описі веб-сервісів.

Математичними підвалинами мов опису онтологій (зокрема, OIL, DAML+OIL, OWL) є логіка. Коротко розглянемо деякі логіки – формальні мови із добре визначеною семантикою, які є придатними для рішення задач формалізації і машинної обробки знань.

Логіка висловлювань є однією з найпростіших формальних систем $L = L(A, \Omega, Z, I)$. Її складають: скінченна множина A атомарних висловлювань, які можуть бути або істинними, або хибними; скінченна множина $\Omega = \cup\Omega_j$ логічних зв'язок з підмножин Ω_j арності $j = 0..m$; скінченна множина правил виводу Z та скінченна множина аксіом I. Зазвичай використовують $\Omega_0 = \{ 0, 1 \}$; $\Omega_1 = \{ \neg \}$ (унарна операція заперечення); $\Omega_2 = \{ \wedge, \vee, \Rightarrow, \Leftrightarrow \}$ (відповідно, бінарні операції кон'юнкції, диз'юнкції, односторонньої та двосторонньої імплікації). За допомогою атомарних висловлювань та зв'язок рекурсивно будуються складні висловлювання (формули системи L).

Проаналізуемо простий, приклад з області грід-обчислень (детальніше грід та грід-сервіси розглядаються у наступному підрозділі). Нехай маємо множину користувачів грід, кожен з яких є членом як мінімум однієї віртуальної організації (ВО). Нехай є множина ресурсів Грід, причому доступ до кожного з ресурсів дозволено лише обраним віртуальним організаціям. Задача: зробити висновок щодо задоволення запиту певного користувача на використання певного ресурсу.

Визначимо висловлювання типу «i-й користувач є членом у j-ій ВО» як $User_i IsMemberOfVO_j$, висловлювання «j-та ВО має доступ до k-го ресурсу» як $VO_j CanAccessResource_k$,а висловлювання «i-й користувач має доступ до k-го ресурсу» як $User_i CanAccessResource_k$. База знань (БЗ) системи міститиме записи про приналежність користувачів до ВО, про доступ ВО до ресурсів, і про правила прийняття рішення щодо дозволу користувача працювати на ресурсі. Наприклад, такі:

$User_1IsMemberOfVO_1 \wedge \neg User_1IsMemberOfVO_2 \wedge ...,$

(користувач №1 є учасником ВО №1, але не ВО №2, і т.д.)

$User_2IsMemberOfVO_1 \wedge User_2IsMemberOfVO_2 \wedge ...,$

(користувач №2 є учасником і ВО №1, і ВО №2, і т.д.)

...

$VO_1CanAccessResource_1 \wedge VO_1CanAccessResource_2 \wedge ...,$

(ВО №1 має доступ і до ресурсу №1, і ресурсу №2, і т.д.)

$\neg VO_2CanAccessResource_1 \wedge VO_2CanAccessResource_2 \wedge ...,$

(ВО №2 не має доступу до ресурсу №1, але має доступ до ресурсу №2, і т.д.)

...

$(User_1IsMemberOfVO_1 \wedge VO_1CanAccessResource_1) \vee ($

$User_1IsMemberOfVO_2 \wedge VO_2CanAccessResource_1) \vee ... \Rightarrow$

$User_1CanAccessResource_1$,

$(User_1IsMemberOfVO_1 \wedge VO_1CanAccessResource_2) \vee ($

$User_1IsMemberOfVO_2 \wedge VO_2CanAccessResource_2) \vee ... \Rightarrow$

$User_1CanAccessResource_2$,

...

$(User_2IsMemberOfVO_1 \wedge VO_1CanAccessResource_1) \vee ($

$User_2IsMemberOfVO_2 \wedge VO_2CanAccessResource_1) \vee ... \Rightarrow$

$User_2CanAccessResource_1$,

...

Використовуючи відомі алгоритми для виведення значення висловлювання $User_1CanAccessResource_1$ можна отримати відповідь, чи може користувач №1 отримати доступ до ресурсу №2. Цей найпростіший приклад показує, що логіка висловлювань (класична логіка нульового порядку) є придатною для опису фактів та логічних зв’язків у простих середовищах.

Якщо кількість користувачів ресурсів буде зростати, то кількість однотипних імплікацій буде складати $m \times n$, де m – кількість користувачів, n – кількість ресурсів, а кількість диз'юнктів у кожній такій формулі дорівнюватиме кількості ВО. Потенційно велики значення m та n обумовлюють велику кількість однотипних записів у БЗ для вираження правила – доступ до конкретного ресурсу має будь-який користувач, що є членом будь-якої ВО, яка має доступ до цього ресурсу. Якщо ж врахувати, що особливістю Грід є динамічна структура (за одним з визначень – динамічна множина ресурсів та користувачів), то очевидно, що логіці висловлювань бракує потужності для того, щоб стисло виразити знання про складні варіанти середовища, яке розглядається.

У якості інших прикладів можна навести наступні класичні міркування (на тематику Грід), які неможливо виразити логікою висловлювань: «будь-який учасник ВО №1 є учасником ВО №2; користувач Х – не учасник ВО №2; значить, він не учасник ВО №1», «в усіх людей є прізвище; усі користувачі – люди; значить, в будь-якого користувача є прізвище», «оперативна пам'ять – це апаратне забезпечення; значить, вимоги до пам'яті – це вимоги до апаратного забезпечення».

Розвитком логіки висловлювань (логіки нульового порядку) є логіка предикатів (логіка першого порядку, англ. first-order logic, FOL). Її мову складають множини функціональних F та предикатних P символів певної арності. Додатково використовуються символи змінних, пропозиціональні зв'язки (ті ж, що і для логіки висловлювань), квантори всезагальності $\forall$ та існування $\exists$. На базі цього алфавіту будуються складніші конструкції: терми, атоми, формули.

Терм – це символ змінної, що має вигляд $f(t_1, \ldots, t_n)$,

де f – функціональний символ, $t_1, \ldots, t_n$ – терми.

Атом має вигляд $p(t_1, \ldots, t_n)$, де p – предикатний символ.

Формула – це атом, що описується $\neg F$, $F_1 \vee F_2$, $F_1 \wedge F_2$, $F_1 \Rightarrow F_2$, $\forall x\, F$, $\exists x\, F$, де F, F_1, F_2 -формули, а x – символ змінної.

Отже дана логіка, на відміну від попередньо розглянутої, має справу не просто з фактами, а з об'єктами, відношеннями, функціями.

Розглянемо задачу дозволу доступу користувача до ресурсу.

Нехай відношення *IsMemberOf* означає «є учасником», *CanAccess* – «має доступ до». Тоді мовою логіки першого порядку правило дозволу доступу можна узагальнити так:

$$\forall x,y,z\ (\ IsMemberOf\ (\ x,\ y\ ) \wedge CanAccess\ (\ y,\ z\ )) \Rightarrow CanAccess\ (\ x,\ z\ ).$$

Виразна потужність логіки першого порядку складає деякі проблеми для операцій логічного виводу: немає гарантій щодо здійсненності виводу за скінченний час. Тому за основу для мови опису онтологій OWL була взята більш проста варіація FOL – описова (дескрипційна) логіка. Логіки цієї родини є компромісом між виразністю та обчислювальністю. Описові логіки оперують поняттями «концепт» (одномісний предикат, клас) та «роль/властивість» (двомісний предикат, бінарне відношення). Можна сказати, що описові логіки є «варіантом FOL з двома змінними», тобто не більше двох змінних під кванторами, що є decidable. Розрізняють твердження: загального вигляду (стосуються класів, т.зв. . Логіки оперують двома твердженнями: загального вигляду – «блок термінів» T-Box (terminological box), та конкретного вигляду – «блок тверджень» A-Box (assertion box). Разом ці групи тверджень формують базу знань (БЗ).

Синтаксис описової логіки складають символи атомарних концептів, ролей та конструкторів до числа яких входять логічні зв'язки FOL за допомогою котрих рекурсивно будуються складні концепти.

Значна кількість логік відрізняються множиною конструкторів. Якщо логіка використовує конструктори наведені в таблиці 3.1, то до її найменування додається відповідна літера.

Так, наприклад, логіка SHOIN(D) розшифровується так. S = ALC + R_+, тобто дозволене об'єднання, перетин, квантори (AL), а також заперечення атомів (C) та транзитивні ролі (R_+). Дозволені також: ієрархія ролей (H), номінали (O), інверсні ролі (I), числові обмеження (N), обмеження по типам даних (D).

Таблиця 3.1.Позначення описових логік

Літери				Конструктор	
S	ALC	FL-		A	Концепт
				R	Ім'я ролі
				C ⊓ D	Перетин
				∀R.C	Обмеження значення
				∃R	Квантор існування
		AL*		⊤⊥	Концепти верхнього та нижнього рівня
			U	C ⊔ D	Об'єднання
			C	¬C	Заперечення концептів
			A	¬A	Заперечення атомів
			E	∃R.C	Обмеження на існування
			O	$\{a_1,\ldots,a_n\}$	Номінали (концепт як множина індивідів)
			N	(≥ n R) (≤ n R)	Числові обмеження
			Q	(≥ n R.C) (≤ n R.C)	Числові кваліфіковані обмеження
S				R_+	Транзитивні ролі
			I	R^-	Інверсні ролі
			H	R ⊆ C	Ієрархія ролей
			R		Розширення ієрархії ролей до композиції ролей
			F		Обмеження на функціональність ролей
			D		Обмеження на типи даних

Перш ніж перейти безпосередньо до розгляду найпоширенішої у веб-просторі мови опису онтологій OWL, варто згадати одне можливе доповнення до логіки SHOIN(D), яка є основою OWL-DL, – правила, що базуються на диз'юнктах Хорна.

Диз'юнкти Хорна є диз'юнкцією літералів (нульового, першого порядку), з яких не більше одного є додатніми (без заперечення). З цього визначення виплаває, що кожен хорнівський вираз може бути записаний як імплікація додатніх літералів. Це значно спрощує читання та створення формальних висловлювань людьми, а процедура логічного виводу є достатньо ефективною. Мова правил на базі хорнівських виразів покладена в основу мови Prolog. Мова SWRL (Semantic Web Rule Language), яка також є представником цього сімейства, позиціонується як можливе доповнення до описової логіки при створенні БЗ. У деяких випадках використання SWRL виправдане лише як засоб для покращення читабельності, однак правила з більш ніж одною спільною змінною у передумові та наслідку правила неможливо передати описовою логікою.

Здійснення логічних виведень з БЗ на описовій логіці є одною з найцінніших можливостей, яку надає використання онтологій. Зазвичай для цього будуються окремі незалежні програмні модулі (Reasoners).

Розроблено стандартний інтерфейс доступу до таких компонентів логічного виведення – DIG Interface. Втім, його виразних можливостей недостатньо для перекладу онтологій на OWL-DL. Коротка характеристика деяких найбільш відомих модулів логічного виводу наведена в табл. 3.2.

До базових задач модулів логічного виводу відносять такі:

- перевірка несуперечливості онтології (онтологія є несуперечливою, якщо для неї існує хоча б одна модель);
- перевірка задовільності окремого концепту/групи концептів (концепт задовільний, якщо можливо знайти інтерпретацію, при якій він не перетворюється на порожню множину);
- перевірка відношення підлеглості (категоризація) між двома концептами;
- класифікація цілої онтології – створення таксономії.

Таблиця 3.2. Модулі логічного виведення (різонери)

Назва	**Ліцензія**	**Логіка**	**Мови, інтерфейс**	**Платформа**
CEL	Free	EL+	KRSS, DIG	Lisp
Cerebra Engine	$		OWL-API	C++
FaCT++	Free - (L)GPL	SROIQ	OWL-API, Lisp-API, DIG	C++, open-source
fuzzyDL	Free	SHIF		Java / C++
HermiT	Free - LGPL	SHIQ	KAON2 API	Java
KAON2	Free	SHIQ +SWRL	Власний інтерфейс, DIG	Java
MSPASS	Free	ALB		C, open-source
Pellet	Free	SROIQ	OWL-API, DIG, Jena API	Java, open-source
QuOnto	Free	DL-Lite	Власний інт-с	Java
RacerPro	$	SHIQ	OWL-API, DIG	Lisp
SHER	$	SHIN		Java

Імовірно, варто очікувати у перспективі злиття існуючого різноманіття підходів до опису веб-сервісів у деякий об’єднаний стандарт, подібно до історії з мовою OWL. На даний момент, існуючі технології (UDDI, SWS, OWL-S та ін.) є занадто складними для реалізації та

поширення, а багато з них вже відмирають. Тому для репозитарію сервісів доцільно розробити власну онтологію (OWL, RDF), що описує основні характеристики сервісу: входи, виходи, формати даних, призначення (для семантичного пошуку користувачем або програмним агентом), місцезнаходження. Це дозволило б здійснювати простий пошук сервісів під кінцеву мету компонування з урахуванням наявних даних. Щодо існуючих реалізацій семантичних інструментів, то поєднання Apache Jena та серверу Joseki дасть можливість ставити SPARQL-запити до OWL/RDF-бази знань про сервіси.

3.3. Грід-сервіси та їх оркестрування

Грід-інфраструктура --- ресурси, виділені у спільне використання, та спеціалізоване службове програмне забезпечення проміжного шару або рівня (ПЗПШ або ПЗПР) --- представляє собою універсальний інструмент для проведення масштабних розрахунків. Універсальність його полягає в теоретичній можливості рішення обчислювальних задач довільної складності в адекватний час на розподілених ресурсах.

Грід-обчислення в тому вигляді, в якому вони існують нині, з’явилися трохи більше ніж десять років тому, хоч ідея об’єднання високопродуктивних ресурсів у віртуальний суперкомп’ютер набагато давніша. За відносно нетривалий час розвитку грід-обчислень теоретиками грід було сформульовано безліч визначень того, що може вважатися грід-системою. Ми ж просто надамо визначення грід-системі через перелік її властивостей.

Спільний доступ до розподілених неоднорідних ресурсів. Унікальність грід в тому, що ця технологія надає доступ до ресурсів, рознесених географічно, належних різним власникам, без централізованого обслуговування та адміністрування, з різною

архітектурою та ОС для користувачів, які не мають облікових записів на цих ресурсах. Натомість, доступ надається за приналежністю до так званих віртуальних організацій (ВО). Перевага: простота доступу до будь-якого з ресурсів грід. Недолік: відсутність контролю за ресурсами не дозволяє сподіватися на негайне виконання обчислень в разі високої зайнятості ресурсів.

Об'єднання обчислювальних потужностей. З точки зору користувача грід, він має доступ до "віртуального суперкомп'ютера" з сумарними апаратними ресурсами значно більшими, ніж у будь-якого окремо взятого комп'ютера, що дозволяє розглядати грід-мережу ресурсів як реальну дешеву альтернативу власному виділеному кластеру. Перевага: можливість проводити масштабні обчислення, займаючи вільні ресурси кількох машин. Недолік: віддалені ресурси знаходяться у спільному доступі для усіх користувачів ВО, а тому можуть бути повністю завантажені.

Динамічний пул ресурсів. Грід-система складається з динамічної множини ресурсів, що можуть як бути додані до системи, так і бути виведені з неї в будь-який момент їх власниками. Перевага: нарощення потужності відбувається без переривання роботи грід-системи. Недолік: немає гарантії в успішному завершенні обчислень на тому ресурсі, на якому вони стартували.

Безпека. Специфіка організації доступу до ресурсів для усіх членів ВО обумовила появу так званої інфраструктури безпеки грід, заснованої на використанні x.509-сертифікатів для процедур аутентифікації та авторизації. Перевага: наявність дійсного грід-сертифікату та членство у ВО дозволяють отримати доступ до будь-якого ресурсу цієї ВО. Недолік: цілий ряд незручностей для користувачів, пов'язаних з процедурою отримання та використання сертифікату, необхідністю забезпечення його надійного зберігання, що обмежує користувачів в способах доступу до грід-систем.

Проміжний програмний рівень. Забезпечення прозорого для користувача доступу до об'єднаних гетерогенних ресурсів, координація інформаційної системи та інші функції з підтримки функціонування грід покладені на так зване програмне забезпечення проміжного рівня (ПЗПР) грід. Можна виділити основні підсистеми, спільні для більшості ПЗПР:

- керування виконанням завдань та планування навантаження;
- керування зберіганням та передачею даних;
- інформаційна система;
- інфраструктура безпеки.

Перевага: ПЗПР надають користувачу готовий інтерфейс з необхідним мінімумом функцій, достатнім для роботи в грід, приховуючи при цьому складність архітектури розподіленої системи. Недолік: відсутність стандартів робить різні ПЗПР несумісними між собою, що породжує ряд проблем та незручностей як для користувачів, так і для розробників ПЗ для грід.

Серед існуючих ПЗПР, що активно використовуються в масштабних грід-проектах, можна виділити наступні: Globus Toolkit 2, 4, 5, Unicore 6, EGEE/EMI gLite 3, Nordugrid Advanced Resource Connector (ARC)0.x/1x/2.x. Останні два дистрибутиви становлять найбільший інтерес для вітчизняних розробок, оскільки використовуються в українській національній грід-інфраструктурі. Орієнтація на виконання поодиноких задач та слабка сумісність різних ПЗПР ускладнює їх використання в комплексах прикладних розрахунків та вимагає розробки ПЗ додаткового рівня, що б відокремило логіку виконання сценаріїв обчислень від специфіки конкретного ПЗПР.

Підсумовуючи вище сказане, визначаємо що грід-інфраструктура це ресурси, виділені у спільне використання, та спеціалізоване службове програмне забезпечення проміжного рівня і представляє собою потужний та досить універсальний інструмент для проведення наукових та інженерних

розрахунків, що потребують значної кількості процесорного часу чи інших ресурсів, яких недостатньо на окремо взятій машині.

Між тим варто зважати на те, що грід не повинен розглядатися як автоматичний прискорювач обчислень, оскільки:

а) для досягнення прискорення виконання на окремому грід-ресурсі необхідно розпаралелити програму для запуску на багатопроцесорному вузлі та/або розподілити дані для обробки між кількома копіями програми, що не здійснюється автоматично ПЗПР;

б) у випадку завантаженості системи час очікування вільних ресурсів є невизначеним.

Грід дає можливість виконання масштабних обчислень, яким потрібні додаткові ресурси, однак не гарантує точного часу їх виконання.

Оркестрування грід-сервісів

Для побудови архітектур, узгоджених із специфікою грід-систем можуть знадобитися сервіси із підтримкою стану (як правило, веб-сервіси не зберігають «стан» між викликами для кращої масштабованості системи). Одним з останніх стандартів з управління станом веб-сервісів є WSRF.

WSRF - це ряд специфікацій, які визначають стандартні «шаблони обміну повідомленнями» для ресурсів веб-сервісів, або, інакше, способи запитів певних властивостей (англ. property) ресурсів або способи вказівки того, що ці властивості повинні бути змінені. Більш того, WSRF визначає стандартні способи вирішення й інших проблемних питань роботи з WS-ресурсами. Сюди входять прості завдання роботи з окремими властивостями, але також і завдання їх групування для таких цілей, як аутентифікація, чи гарантування їх своєчасного знищення.

Тим не менш, не зважаючи на специфіку грід, WSRF-сервіси також потенційно придатні для оркестрування з використанням стандарту WS-BPEL, напряму чи за допомогою традиційних сервісів-обгорток.

3.4. Хмарні сервіси

Хмарні обчислення є новою парадигмою гнучкого споживання і надання обчислювальних сервісів та ресурсів за запитом: IaaS (інфраструктура як сервіс), PaaS (платформа як сервіс) і SaaS (програмне забезпечення як сервіс). Представлення програмного забезпечення у вигляді набору розподілених сервісів, які можуть бути налаштовані і передавати дані між собою, допомагає вирішити проблеми повторного використання програмного забезпечення, проблеми на етапі впровадження та розвитку програм. Таким чином, хмарні обчислення надають можливість ефективної організації сервісно-орієнтованого комп'ютингу з підтримкою функціональної сумісності. В основі хмарних обчислень лежить віртуалізація комп'ютерних ресурсів --- створення віртуальних машин на вільних ресурсах множини реальних серверів. Користувачі можуть арендувати віртуальну інфраструктуру (процесори, пам'ять --- модель IaaS), або готову платформу (ОС, засоби розробки тощо --- модель PaaS), або ж споживати сервіси, сплачуючи за звернення до них (модель SaaS).

Нині існує кілька визнаних провайдерів хмарних послуг, які дають можливість розгортати власні сервіси у хмарі --- Google, Amazon, Microsoft та ін. Надалі такі «хмарні» сервіси можуть на рівних з іншими виступати в (бізнес-) процесах компонування сервісів.

Останнім часом, концепція СОА дістала деякого переосмислення на основі накопиченого досвіду, та оформилася у вигляді концепції **«мікросервісів».** Концепція мікросервісів (рис.3.2) базується на наступних засадах:

- чітка функціональна декомпозиція за принципом "одна задача — один мікросервіс";
- зосередження логіки в кінцевих точках (сервісах), а не в каналах передачі даних (простиставляється СОА-патерну ESB –"шині"

обміну повідомленнями між сервісами, відповідальній за узгоження численних інформаційних протоколів та форматів повідомлень, оркестрування сервісів, часто навантаженій й іншими функціями, відмінними від простої передачі даних від постачальників до споживачів);

- децентралізоване управління даними (жодних спільних централізованих БД);
- проектування “під відмову” (окремі мікросервіси мають зберігати працездатність в умовах імовірних відмов інших сервісів екосистеми);
- асинхронна взаємодія (мінімізує простоювання сервісів в очікуванні відповіді на їх запити).

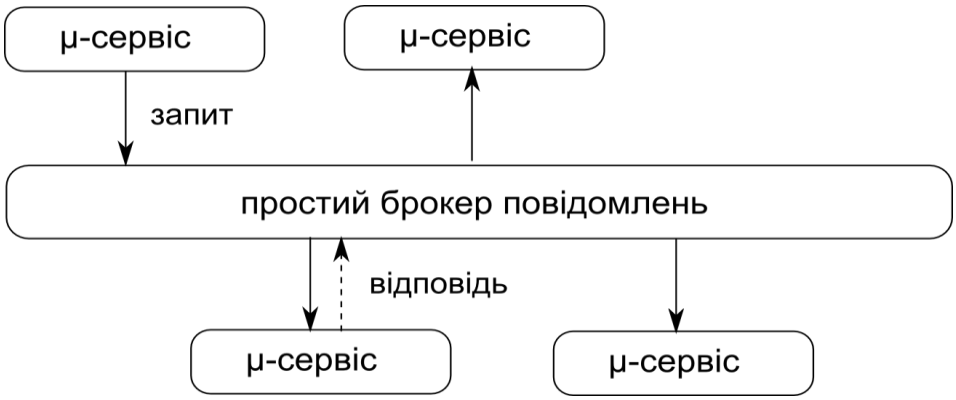

Рис.3.2. Мікросервісна архітектура з асинхронним обміном повідомленнями через простий канал

Можливості, які надають СОА та веб-сервіси, надалі стимулювали появу концепції «*безсерверної архітектури*» (англ. Serverless architecture). Її суть в тому, що для програмних клієнтів (веб-клієнтів, мобільних додатків тощо) не існує поняття єдиного сервера, до якого вони звертаються. Натомість існує множина (або хмара) серверів, які надають певний функціонал.

Ще більшою є різниця для розробників або власників програмного забезпечення – на відміну від традиційних хмар типу IaaS, PaaS чи SaaS (детальніше – у наступних розділах), вони мають справу з новим типом хмар – FaaS (англ. *Function as a Service*), де відповідальність за керування віртуальними машинами та обліком спожитиж ресурсів повністю несе «хмарний» провайдер, а не споживач ресурсів. Користувачі хмар платять за факт виклику процедури (постачаються як ті ж веб-сервіси), а не за спожиті ресурси. Прикладами такого підходу є Amazon Web Services Lambda та Google Functions.

3.5. Розробка онтології сервісів

Розробку онтології найбільш зручно виконувати у спеціалізованих середовищах, аналогічно до середовищ розробки веб-сервісів. Одним з провідних засобів тут є редактор Protégé (рис.3.3). Причому доступна і його онлайн-версія.

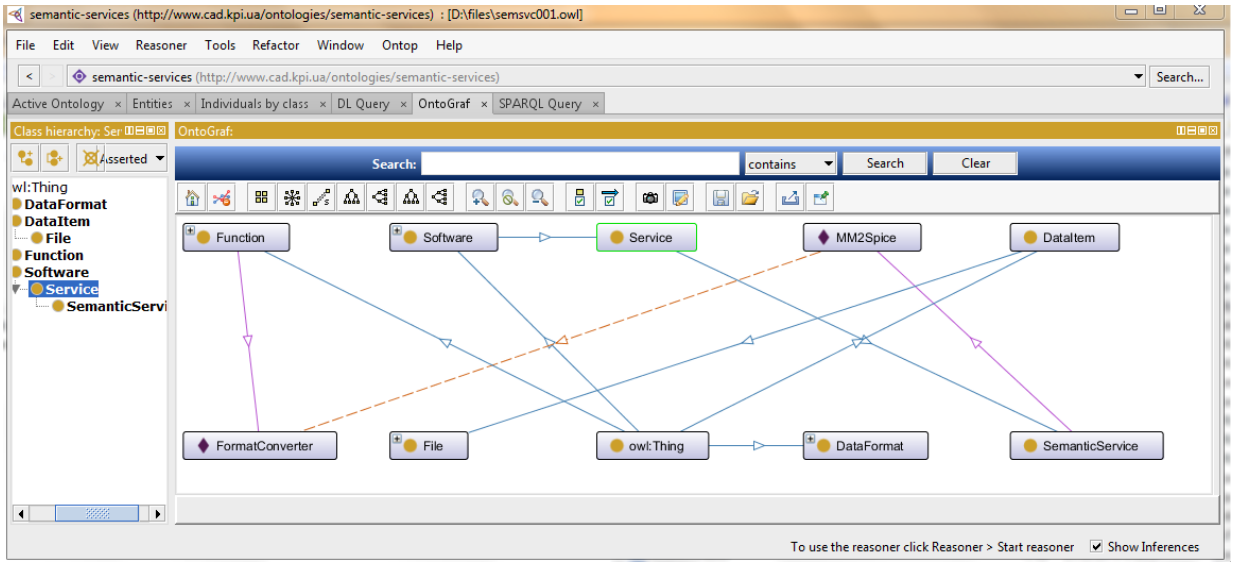

Рис.3.3. Інтефейс користувача редактора Protégé

Почати роботу варто з розробки базових понять – сервіс, функція, входи, виходи, формат даних, адреса, порт тощо. Після чого можна розширювати онтологію додатковими зв’язками та фактами про наявні

сервіси. Редактор також має вбудований інтерфейс для SPARQL-запитів, щоправда, він не виконує логічних виводів нових фактів, для чого слід залучати вбудовані «різонери», а потім завантажувати нову, розширену онтологію. Або ж скористатися зв'язкою засобів Jena + сервер Joseki, для побудови повноцінного веб-сервісу реєстру.

3.6. Розгортання сервісів у хмарі

Серед різноманіття постачальників послуг хостингу коду (веб-сервісів) можна звернути увагу на Heroku (рис.). Цей сервіс надає можливість безкоштовного хостингу початового рівня, причому розгортання Java-веб-сервісів в ньому здійснюється відносно просто.

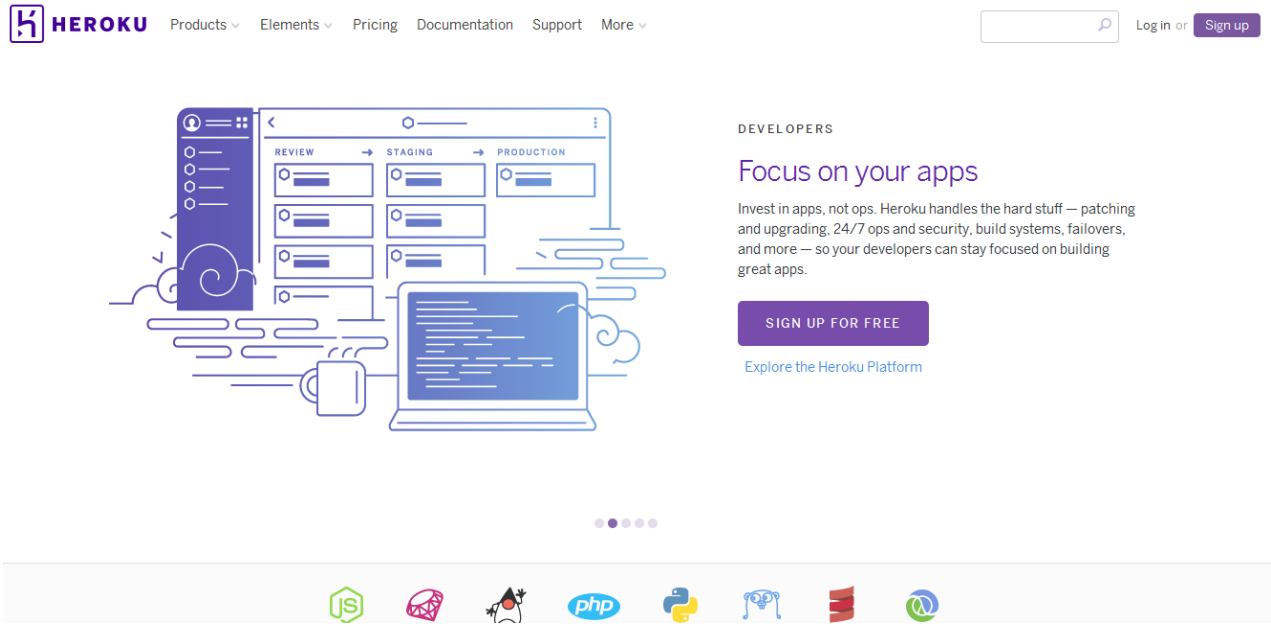

Рис.3.4. Веб-інтерфейс порталу доступу Heroku

Перший крок – запуск Webapp Runner, що запустить сервер Tomcat на будь-якій машині, де встановлено Java:

```
java -jar webapp-runner.jar application.war
```

Далі --- створення Maven-додатку:

```
mvn archetype:generate -DarchetypeArtifactId=maven-archetype-webapp
```

Далі – збірка додатку:

```
mvn package
```

І, нарешті, – запуск додатку:

```
java -jar target/dependency/webapp-runner.jar target/*.war
```

Після тестування можна розгортати додаток у хмарі. Для цього слід створити профіль:

```
web:    java $JAVA_OPTS -jar target/dependency/webapp-runner.jar --port
$PORT target/*.war
```

Потім --- коміт коду до хмарного репозиторію:

```
git init
git add .
git commit -m "Ready to deploy"
```

Далі створення хмарного додатку:

```
heroku create
```

І розгортання додатку:

```
git push heroku master
```

Більше інформації доступно на сайті Heroku:
https://devcenter.heroku.com/articles/java-webapp-runner

3.7. Завдання для самостійної роботи

1. Спробуйте побудувати онтологію сервісів фільтрації зображень та вивести з неї факти за допомогою модулю логічного виведення. Перевірте роботу через SPARQL-запити.

2. Спробуйте побудувати сервіс-реєстр, що здійснював би запити до бази знань про розроблені сервіси з метою пошуку сервісів за їх призначенням.

2. Спробуйте перенести у хмарну інфраструктуру розроблені раніше сервіси, наприклад, сервіси фільтрації даних.

Висновки по розділу

1. Проблема автоматизації пошуку веб-сервісів, особливо при залученні сервісів сторонніх постачальників, залишається актуальною та може бути вирішена за рахунок семантичних реєстрів.

2. Веб-сервіси нині успішно використовуються як універсальний механізм організації доступу до унікальних ресурсів (таких, як грід-інфраструтура та хмарні ресурси). Втім, слід пам’ятати про особливості роботи з цими типами ресурсів (в першу чергу --- про механізми аутентифікації та авторизації, що мають бути дотримані при виконання бізнес-процесів із залученням таких сервісів).

3. Загалом, концепція СОА та хмарні обчислення разом продовжують розвиватися і пропонують все більше гнучкості для розробників програмного забезпечення.

Контрольні питання

1. Що таке реєстр сервісів? Яка його структура?
2. Що включає в себе поняття «семантичні технології»?
3. Що таке онтологія, як її описати?
4. Що таке грід-обчислення та грід-сервіси? Яка їх специфіка?
5. Як пов’язані хмарні обчислення та СОА?

Перелік умовних позначень

BPEL	-	мова опису виконання бізнес-процесів
BPMN	-	графічна нотація для опису бізнес-процесів
СКБП	-	система керування бізнес-процесами
HTTP(S)	-	протокол передачі гіпертексту (захищений)
IaaS, PaaS, SaaS	-	рівні хмарних послуг: інфраструктура, платформа та програмне забезпечення як сервіс відподвіно
JSON	-	формат даних, аналог XML
ЛСКР	-	локальна система керування ресурсами
OGSA	-	відкрита архітектура грід-сервісів
OWL	-	мова опису онтологій у веб
REST	-	протокол доступу до веб-сервісів без стану
RDF	-	система опису семантичних триплетів
RIA	-	розвинений інтернет-додаток
RMI	-	віддалений виклик методів
RPC	-	віддалений виклик процедур
SOA, SOC, COA	-	сервісно-орієнтована архітектура (або обчислення)
SOAP	-	простий протокол доступу до об’єктів
SPARQL	-	мова запиту до бази знань
UDDI	-	інструмент для публікації описів веб-сервісів
WfMS, СКП, СКПР	-	система керування потоками робіт
WSDL	-	стандартна мова опису веб-сервісів
WSRF	-	платформа ресурсів веб-сервісів
XML	-	розширювана мова розмітки
ПЗПШ, M/W	-	програмне забезпечення проміжного шару у грід

Перелік літератури

1. Erl T. Service-Oriented Architecture: Concepts, Technology and Design. – New York: Prentice Hall/PearsonPTR. —- 2005. — 792 p.
2. Newcomer E. Understanding Web Services: XML, WSDL, SOAP, and UDDI. — Addison-Wesley, USA. — 2002. – 368 p.
3. Foster I. What is the Grid? A Three Point Checklist // Argonne National Laboratory, University of Chicago, Grid Today.—- 2002.—- Vol.1, № 6. — p.32-36.
4. Петренко А.І. Семантичний Грід для науки і освіти / Петренко А.І., Булах В.В., Хондар В.С. – К. : НТУУ “КПІ”, 2010. – 180 с.
5. Булах Б.В. BPEL-орієнтована система управління інженерними та науковими обчислювальними сценаріями / Булах Б.В., Петренко А.І. // Вісник університету «Україна»: Інформатика, обчислювальна техніка та кібернетика. — К.:Університет «Україна».— 2011.— № 2. — С.90-100.
6. Петренко А.І. Застосування workflow-систем для потреб сучасних науки та інженерії / Петренко А.І., Булах Б.В. // Наукові вісті НТУУ «КПІ».-К.: «Політехніка».— 2011. — №5(79). — с.40-51.
7. Булах Б.В. Підхід до компонування rest-сервісів для виконання інженерних обчислень / Булах Б.В., Яременко В.С. // Міжнародний науковий журнал “ScienceRise” . – Х.: “Технологический Центр” – 2015. – т.7, №2 (12). – С. 9-14.
8. Slominski A. Adapting BPEL to Scientific Workflows // Chapter 14 in I.J. Taylor, E. Deelman, D.B. Gannon, M. Shields (Eds.). Workflows for e-Science. Scientific Workflows for Grids. — Springer. — 2006. — p. 208-226.
9. Веб-сторінка Triana —- Open Source Problem Solving Software — Режим доступу: http://www.trianacode.org/

10. Веб-сторінка Kepler Project — Режим доступу: https://kepler-project.org/

11. Веб-сторінка Taverna Workflow Management System — Режим доступу: http://www.taverna.org.uk/

12. Веб-сторінка Web Service Definition Language (WSDL) — Режим доступу: www.w3.org/TR/wsdl

13. Веб-сторінка Unified Service Description Language — Режим доступу: www.w3.org/2005/Incubator/usdl

14. Веб-сторінка Web Services Business Process Execution Language — Режим доступу: docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

15. Веб-сторінка Web Services Choreography Description Language Version 1.0 — Режим доступу: http://www.w3.org/TR/ws-cdl-10