View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ISS Library

D2.4.6 – A Theoretical Integration of Web Service Discovery and Composition

Теоретична інтеграція виявлення і композиції Веб-сервісів

Резюме

У даному документі розглядається проблема забезпечення комплексного підходу до автоматичного виявлення і композиції доступних семантичних Веб-сервісів для того, щоб виконати даний запит користувача, вказаний в якості мети композиції. Це вимагає інтеграції різних функціональностей, а саме виявлення Веб-сервісу, а також двох рівнів композиції сервісу: (I) композиція на функціональному рівні і (II) композиція на рівні процесу. Кожна з цих функціональних можливостей вже досліджувалась і застосовувалась в ізоляції до семантичних Веб-сервісів (див. Deliverable 2.4.2), але їх інтегроване використання є, як і раніше, відкритою проблемою.

Проблему автоматичного виявлення сервісів можна розглядати як задачу локалізації сервісу, який може виконувати деякі завдання запитуючої сторони. Композиція функціонального рівня розширює проблему виявлення у разі, якщо не представляється можливим знайти жодного сервісу, який може виконати мету композиції. В композиції функціонального рівня набір існуючих сервісів, які обрані, поєднані підходящим способом, можуть спільно виконувати мету композиції. Композиція на рівні процесу охоплює останній етап загальної задачі композиції і дозволяє побудувати виконуваний Веб-сервіс, який реалізує композицію. На цьому етапі ми припускаємо, що набір Веб-сервісів, необхідних для визначення композиції, вже знайдений і, що ми повинні опрацювати деталі про те, як взаємодіяти з ними.

У цьому документі пропонуються теоретичні засади і програмна архітектура для вирішення проблеми інтеграції виявлення і композиції. Точніше, він надає приклад опису сценарію, де необхідно інтегрувати виявлення і композицію, щоб відповідати вимозі замовника. Він досліджує існуючі мови і основні теоретичні моделі, необхідні для визначення ключових елементів проблем (існуючі сервіси, мета композиції і виконуваний композитний сервіс). Нарешті, він пропонує архітектуру програмного забезпечення для інтеграції виявлення і композиції.

ЗМІСТ

1 Вступ....3
1.1 Огляд документу....4
2 Фонові роботи....5
2.1 Виявлення сервісу....5
2.2 Композиція функціонального рівня....5
2.3 Композиція рівня процесу....6
3 Мови для інтеграції виявлення і композиції....8
3.1 Сценарій віртуального туристичного агентства (Virtual Travel Agency Scenario)....8
3.2 Моделювання Веб-сервісів....10
3.2.1 Підхід WSMO....10
3.2.2 Підхід BPEL4WS....19
3.3 Моделювання мети композиції....22
3.4 Моделювання композитного сервісу....23
4 Теоретичні моделі....25
4.1 Моделі для онтології предметної області....25
4.2 Моделі для веб-сервісів....25
4.3 Моделі для мети композиції....27
4.4 Моделі для композитного сервісу....28
5 Архітектура для інтеграції виявлення і композиції....29
5.1 Огляд архітектури....29
5.2 Визначення інтерфейсів....30
5.3 Основні функціональності....32
6 Висновки....34
6.1 Розширення архітектури: репутація....34
Література....36

1 Вступ

У даному документі розглядаються проблеми забезпечення комплексного підходу до автоматичного виявлення, вибору і композиції доступних семантичних Веб-сервісів у новий виконуваний Веб-сервіс, який відповідає даному запиту користувача, визначений як мета композиції. Генерація композитного виконуваного процесу вимагає інтеграції та гармонізації різних існуючих функціональностей, доступних для семантичних Веб-сервісів, таких як виявлення, композиція на функціональному рівні і композиція на рівні процесу. Наша мета полягає в тому, щоб запропонувати теоретичне рішення і програмну архітектуру, які побудовані на вершині цих компонентів, і що використовує їх у комбінованому ітеративному підході для того, щоб побудувати композитний виконуваний сервіс.

Автоматизована композиція Веб-сервісів є однією з найбільш перспективних ідей і - у той же час - одна з головних проблем для зльоту сервіс-орієнтованих застосувань: сервіси, які автоматично компонуються, можуть виконувати нові функції, взаємодіючи з сервісами, які публікуються в Інтернеті, таким чином, значно зменшуючи час і зусилля, необхідні для розробки нових веб- і сервіс- орієнтованих додатків. Широке визнання отримало те, що одним з ключових елементів для автоматизованої композиції Веб-сервісів є семантика: однозначні описи можливостей Веб-сервісів і процесів Веб-сервісів (наприклад, в таких мовах, як OWL-S [Coa03] або WSMO [WSM05]), які забезпечують здатність міркувати про Веб-сервіси, і для автоматизації завдань Веб-сервісів, таких як виявлення Веб-сервісу і композиція, див., наприклад, [MSZ01].

Велика частина робіт про композицію семантичних Веб-сервісів була зосереджена поки на проблемі композиції на *функціональному рівні*, тобто композиція шляхом зіставлення передумов і ефектів сервісів, описаних як атомарні компоненти, які, враховуючи деякі входи, повертають деякі виходи [PSK02, CFB04]. Композиція на функціональному рівні, як правило, поєднана з методами виявлення сервісів, які використовуються, щоб знайти екземпляри підходящого сервісу для композиції [CFB04]. Однією з ключових відкритих проблем для семантичних Веб-сервісів є розширення композиції функціонального рівня для того, щоб автоматично генерувати композитні Веб-сервіси, які можуть бути безпосередньо виконані, викликавши компонентні сервіси, для досягнення деякої композитної мети. Це є ключовим кроком у скороченні зусиль, часу і помилок, пов'язаних з ручною композицією на рівні програмування.

Проблема такої повної інтеграції виявлення і композиції є далеко не тривіальною. Ми повинні взяти до уваги той факт, що, в реальних випадках, компонентні сервіси не є атомарними, і не можуть взагалі бути виконані в одному кроці запит-відповідь. Взагалі, кожен компонентний сервіс може бути вказаний як протокол взаємодії, де різні "атомарні" виклики та відповіді були об'єднані в складні шаблони виконання. Хоча деталі точного протоколу, необхідного для взаємодії з існуючим сервісом, не важливі у виявленні авансом, вони стають необхідні, коли ми прагнемо до генерації композитних Веб-сервісів, які є виконувані. З цієї причини, композиція виконуваних сервісів повинна мати справу з описами Веб-сервісів з точки зору складних процесів, які складаються з довільних комбінацій атомних взаємодій, в стилі, наприклад, OWL-S моделей процесів [Coa03] або на основі абстрактної машинної моделі, такі, як у WSMO інтерфейсах [WSM05].

Підхід до наскрізної (end-to-end) інтеграції виявлення і композиції, запропонованої в цій статті, працює в два етапи. На першому етапі, тобто в момент *виявлення* та протягом композиції на *функціональному рівні*, необхідно визначити набір Веб-сервісів, що, взаємодіючи один з одним, може бути в змозі відповідати запиту композиції. У центрі уваги є необхідні входи і надавані виходи сервісів, щоб отримати виходи, необхідні користувачу. Наприклад, на цьому рівні ми виявляємо, що сервіс «бронювання готелів» і сервіс «бронювання авіаквитків», необхідні, щоб задовольнити запит на відпустку користувача. На другому етапі, враховуючи набір обраних Веб-сервісів і з урахуванням мети композиції, композиція *рівня процесу* відповідає за генерування автоматично виконуваного композитного Веб-сервісу. Наприклад, дано моделі процесів двох доступних Веб-сервісів для «бронювання готелів» і «бронювання польоту», ми прагнемо генерувати виконуваний композитний сервіс, скажімо, «віртуальна туристична фірма».

Взаємодіючи з сервісами «бронювання готелів» і «бронювання польоту», композитний сервіс бронює готельні номери та льотні місця відповідно до заданої мети.

У цьому документі обговорюється стандартна теоретична установка для вирішення проблеми інтеграції виявлення і композиції, як описано вище. Крім того, він описує програмну архітектуру, яка повторно використовує існуючі підходи до виявлення, композиції на функціональному рівні і композиції на рівні процесу. Наступним кроком стане розробка прототипу інструменту, заснованого на підході, описаному в цій результуючій роботі.

1.1 Огляд документу

Структура документу виглядає наступним чином. У розділі 2 ми коротко нагадаємо основні поняття виявлення Веб-сервісів, композиції на функціональному рівні та композиції на рівні процесу. Розділ 3 знайомить з мовами, що використовуються для опису різних елементів, що беруть участь у завданні виявлення і композиції. Точніше, ми будемо обговорювати дві мови для визначення компонентних сервісів (WSMO і BPEL4WS), мову для визначення вимог композиції і мову для визначення композитного сервісу (BPEL4WS). У цьому розділі буде використовуватись приклад для ілюстрації різних мов, а також завдання композиції. У розділі 4 ми обговорюємо теоретичну установку, яку ми використовуємо для реалізації інтегрованого завдання виявлення і композиції. Зокрема, ми визначаємо відповідні структури для моделювання елементів композиції, представлених в розділі 3. Розділ 5 пропонує архітектуру для інтеграції виявлення і композиції семантичних Веб-сервісів. Він описує основні програмні компоненти цієї архітектури і інтерфейси між ними. Мета полягає в тому, щоб визначити архітектуру, яка повторно використовує, на скільки можливо, існуючі підходи та алгоритми, щоб дозволити швидку реалізацію інструменту прототипу. Нарешті, у розділі 6 підсумовуються наші результати та плани для майбутніх робіт, і описується, як запропонована архітектура може бути розширена з урахуванням аспектів, пов'язаних з репутацією в задачі композиції.

2 Фонові роботи

У цьому розділі ми коротко нагадаємо поняття виявлення сервісу, композиції функціонального рівня і композиції рівня процесу. Більш детальну інформацію можна знайти в [Lar04a].

2.1 Виявлення сервісу

Виявлення Веб-сервісу є процес пошуку і вибору відповідного Веб-сервісу, який може бути викликаний, щоб відповідати запиту користувача. Виявлення являє собою складний процес, що, в загальному випадку, складається з різних етапів.

Виявлення мети. Починаючи з бажання користувача (виражається за допомогою природної мови або будь-яких інших засобів), виявлення мети буде локалізувати попередньо визначену мету, яка відповідає бажанню запитувача, з набору заздалегідь визначених цілей, отримуючи обрану заздалегідь визначену мету. Така заздалегідь визначена мета є абстракцією бажання запитувача у загальну і багаторазову мету.

Уточнення мети. Обрана заздалегідь визначена мета уточнюється на основі заданого бажання запитувача, щоб насправді відображати таке бажання. Цей крок призведе до формалізованої мети запитувача.

Виявлення сервісу (*Service Discovery*). Виявляють доступні сервіси, які можуть, відповідно до їх абстрактних можливостей, потенційно виконати мету запитувача. Оскільки абстрактна можливість не гарантується бути правильною, ми не можемо гарантувати на цьому рівні, що сервіс буде фактично виконувати мету запитувача.

Укладання договору з сервісом (*Service contracting*). Сервіси, виявлені на основі їх абстрактних можливостей, мають пов'язану можливість укласти договір. Ця можливість укласти договір буде використовуватися при укладанні контрактів з сервісом, щоб визначити, чи обраний сервіс може фактично виконати мету запитувача, встановлюючи контрактну угоду. Якщо це так, то результатом буде сервіс за контрактом (contracted service).

У рамках цього результату, ми зосередимо увагу на *виявленні сервісу* (*Service Discovery*). На цьому етапі ми можемо припустити, що існуючі Веб-сервіси визначаються такими елементами:

• його входи I, тобто значення, які вони отримують при виклику;

• його виходи O, тобто значення, які вони повертають до того, хто їх викликав, у разі успішного виклику;

• попередні умови P, тобто припущення, яких необхідно дотримуватись (на входи і на статус світу) для того, щоб гарантувати успішне виконання сервісу;

• ефекти E, тобто зміни, які відбуваються у разі успішного виконання.

Крім того, мета виявлення може бути визначена подібним набором елементів: у цьому випадку входи і попередні умови виражають значення та умови, які відомі тому, хто викликає, в той час як виходи і ефекти являють собою очікувані результати виклику сервісу.

У цьому контексті, якщо елементи $S = (I, O, P, E)$ визначають сервіс та елементи $g = (I_g, O_g, P_g, E_g)$ визначають ціль виявлення, ми говоримо, що S збігається з g (згідно плагін метчінгу (plug-in matching): див. [Lar04a] для більш докладної інформації), якщо виконані наступні умови:

• $I \sqsubseteq I_g$ і $P \sqsubseteq P_g$ (тобто, входи, запитані сервісами, можуть бути отримані з тих, що відомі інвокеру, і аналогічно для попередніх умов);

• $O_g \sqsubseteq O$ і $E_g \sqsubseteq \mathrm{E}$ (тобто, виходи, запитані інвокером, можуть бути отримані з тих, що повертається сервісом, і так само для ефектів).

2.2 Композиція функціонального рівня

Якщо немає жодного сервісу, який в змозі виконати дану мету (не повний збіг), може бути можливим вибрати набір частково співпадаючих сервісів, які можна композувати у форму робочого процесу для виконання поставленої мети. Ми називаємо процес декомпозиції мети і вибір сервісу *композицією сервісу функціонального рівня*.

Композиція сервісу функціонального рівня вирішує проблему вибору набору сервісів, що, поєднуючись підходящим способом, можуть відповідати даній меті. Кожен існуючий сервіс визначається в термінах атомарної взаємодії, тобто з точки зору його вхідних і вихідних параметрів, і, можливо, також з точки зору його передумов і ефектів. Композиція сервісу функціонального рівня використовує інформацію, яка надається, наприклад, в OWL-S профілі сервісу або в WSMO моделі можливостей сервісу.

Мета визначає загальну функціональність, яку композитний сервіс має реалізувати, знову з точки зору його входів, виходів, передумов і ефектів.

Підхід до композиції сервісу функціонального рівня, запропонований в [Lar04a], заснований на прямій побудові ланцюжка. Неформально, ідея прямої побудови ланцюжка полягає в тому, щоб ітеративно вибрати можливий сервіс *S* і застосувати його до набору вхідних параметрів, які надаються метою *G* (тобто, всі входи, необхідні для *S*, повинні бути доступні). Якщо застосування *S* не вирішує цю проблему (тобто, ще не всі виходи, необхідні для мети *G*, доступні), то нова мета *G'* може бути обчислена з *G* і з виходів, породжених *S*, і в цілому процес повторюється.

Для того, щоб до сервісу *S* можна було застосувати входи, доступні з мети *G*, всі входи, необхідні сервісу *S*, потрібно, щоб відповідали деяким сумісним параметрам у входах, представлених метою *G*. Це означає , що «роль» параметра мети повинна бути такою ж, як, або більш конкретною, ніж, що параметр сервісу, а також діапазон значень, який параметр мети може вжити, повинен бути більш конкретним, ніж прийнятий сервісом *S*.

Після успішної композиції сервісу функціонального рівня, вибрані сервіси розташовуються в робочому процесі, який не порушує дані-залежності між сервісами (наприклад, обмеження на порядок, в якому сервіси можуть бути виконані).

2.3 Композиція рівня процесу

Враховуючи набір існуючих Веб-сервісів W_1, ..., W_n, завдання створення композиції рівня процесу складається із знаходження програми, яка взаємодіє з цими Веб-сервісами у відповідний спосіб, для того, щоб досягти даної вимоги композиції (або мети композиції). Ми називаємо *композицією сервісу рівня процесу* процес створення цієї програми. Розглянемо, наприклад, випадок з Virtual Travel Agency, і припустимо, що набір постачальників туристичних сервісів був визначений для вирішення запиту клієнта. Ці сервіси можуть складається, наприклад, із сервісу бронювання авіарейсу (або сервісу бронювання поїздки на поїзді) та сервісу бронювання готелю, які є достатніми для конкретного запиту замовника, наприклад, конкретний напрям (вибір таких Веб-сервісів може бути результатом композиції на функціональному рівні). Метою композиції на рівні процесу є отримати виконуваний код, що викликає ці Веб-сервіси з тим, щоб отримати пропозицію на запит замовника.

У визначенні виконуваного коду, що реалізує композицію, ми повинні брати до уваги той факт, що в реальних випадках, бронювання готелю не є атомарним кроком, а потребує замість цього послідовності операцій, в тому числі аутентифікації, подання специфічного запиту, узгодження пропозиції, прийняття (або відмова) пропозиції, і бронювання номеру. Тобто, Веб-сервіси W_1, ..., W_n зазвичай композитні, тобто взаємодія з ними не складається з одного кроку запит-відповідь, але вони вимагають слідувати складному протоколу для досягнення необхідного результату. Крім того, кроки, що визначають складну взаємодію не обов'язково визначають послідовність. Справді, ці кроки можуть мати умовний характер, або неномінальні результати (наприклад, аутентифікація може потерпіти невдачу; там може не бути доступної пропозиції від існуючого сервісу...), які впливають на наступні кроки (запит не може бути представлений, якщо не вдається аутентифікація; якщо немає доступної пропозиції, то замовлення не може бути затверджене...). Також може бути випадок, коли та ж сама операція може повторюватися багато разів, наприклад, для уточнення запиту або, щоб обговорити умови пропозиції.

Подробиці про точну послідовність операцій, які необхідні для взаємодії з існуючим сервісом, не є істотними при виявленні сервісу. Взяти ці деталі до уваги стає неминучим, коли виконуваний код, який реалізує композицію, повинен бути згенерований. З цієї причини, в композиції на рівні процесу існуючі Веб-сервіси повинні бути описані в термінах складних, композитних процесів. Ці процеси складаються з довільних (умовних та ітераційних) комбінацій

атомарних взаємодій, і ці атомарні взаємодії можуть мати умовні результати. (Композиція сервісу на рівні процесу використовує інформацію, яка надається, наприклад, в профілі OWL-S сервісу або в моделі WSMO можливостей сервісу). Як наслідок, згенерований виконуваний код повинен бути складною програмою, так як він повинен брати до уваги всі можливі непередбачені обставини, що виникають при взаємодії з Веб-сервісами.

Автоматизована композиція починається з набору Веб-сервісів і з вимоги композиції, і генерує виконуваний Веб-сервіс, який реалізує композитний сервіс. Синтез композитного Веб-сервісу не обмежується атомарними компонентними Веб-сервісами. Вихід цього компонента полягає в тому, щоб визначити протокол взаємодії з вибраними сервісами так, щоб отримати виконувану реалізацію композиції. З цієї точки зору Веб-сервіс визначається як потік (flow) або як протокол взаємодії.

3 Мови для інтеграції виявлення і композиції

У цьому розділі ми введемо деякі стандартні мови, що використовуються для опису різних елементів, які беруть участь у задачі виявлення і композиції. З цією метою ми будемо використовувати приклад, який також дозволить нам визначити вимоги і направити визначення інтегрованого підходу виявлення і композиції. Випадок використання є в контексті сервісів е-туризму, та складається з композиції існуючих сервісів транспорту і розміщення для того, щоб забезпечити сервіс віртуального туристичного агентства для кінцевого користувача. Ми посилаємось на [Lar04b] для отримання додаткової інформації з цього прикладу.

Для моделювання семантичних аспектів сервісів, що беруть участь у віртуальному туристичному агентстві, ми представимо онтології у WSML. Крім того, ми встановимо відношення між цими семантичними описами онтологій домену та поведінковими описами сервісів. Ми покажемо два можливих підходи для цього: з одного боку, використовуючи «класичні» технології Веб-сервісів, де ми виражаємо поведінку, використовуючи BPEL4WS, з іншого боку, ми покажемо, як описати ці поведінкові аспекти спочатку в моделі на основі абстрактних кінцевих автоматів (abstract state machine), заснованої на WSMO. Що стосується останнього, то ми використовуємо синтаксис, заснований на мові WSML [WSM05]. Зверніть увагу, однак, що кілька конструкцій та концептів, які використовуються в цьому розділі, ще не стали частиною офіційного синтаксису WSML. Різні рішення можна отримати, які адаптовані з WSMO групи, коли ці концепти будуть розглянуті виключно в WSML. Крім того, в прикладах ми іноді злегка відхиляємося від WSML синтаксичних обмежень і вимог, якщо це корисно з метою зручності читання.

3.1 Сценарій віртуального туристичного агентства (Virtual Travel Agency Scenario)

Віртуальне туристичне агентство (VTA) є постачальником сервісу е-туризм, який пропонує сервіси бронювання подорожі для кінцевого користувача за допомогою та взаємодіючи з іншими, більш основними постачальниками сервісів електронного туризму. Функціональність VTA є функціональністю традиційного туристичного агентства: отримання запиту від клієнта, вести справу з різними постачальниками електронного туризму, щоб зібрати відповідну пропозицію, яка покриває запит замовника, розміщуючи всі замовлення (та оплати) з різними постачальниками, і прозоро пропонуючи остаточне розміщення поїздки для клієнта. У контексті цього результату ми припускаємо, що доступні провайдери е-туризму повинні бути локалізовані динамічно агентством VTA, без необхідності попередніх домовленостей, і що бізнес процес VTA повинен композуватися динамічно на основі отриманого запиту і доступних постачальників. Надалі ми опишемо варіант використання докладніше.

Мета/Контекст. Клієнт хоче зробити подорож в дане місце (наприклад, Париж) в даний період часу (наприклад, зупиняючись там з 10 серпня по 15 серпня). Клієнт посилає свій запит до VTA, який повинен зібрати пакет, який включає поїздку в/з Париж і проживання протягом всіх ночей, проведених в Парижі. Очевидно, що готель повинен бути замовлений відповідно до польоту (тобто, якщо рейс прибуває 9 серпня, то готель повинен бути заброньований з 9 серпня).

VTA повинен піклуватися про пошук необхідних постачальників сервісів туризму (наприклад, підходящі провайдери польоту для поїздки, готелів у Парижі ...) і зв'язатися з ними. Нарешті, підходяща пропозиція буде повернута клієнтові і на підтвердження або і проживання, і подорож повинні бути замовлені, або ні, що вимагає слабкої форми транзакційності для композитного сервісу.

Актори, що приймають участь.

• Клієнт: кінцевий користувач, який надсилає запит на бронювання подорожі до VTA.

• Постачальники туристичних сервісів: комерційні компанії, які надають конкретні туристичні сервіси.

• VTA: посередник між клієнтом і постачальниками туристичних сервісів.

Вона забезпечує туристичні пакети для клієнтів шляхом об'єднання окремих сервісів різних постачальників туристичних послуг.

Сценарій/кроки.

1. Користувач створює запит на подорож, включаючи всі свої вимоги і переваги.
2. Користувач відправляє запит до VTA.
3. VTA отримує запит і інтерпретує його.
4. VTA вибирає набір постачальників туристичних послуг для задоволення отриманого запиту на подорож.
5. VTA генерує виконуваний код, необхідний для взаємодії з обраними постачальниками туристичних послуг.
6. VTA виконує згенерований код, взаємодіючи з обраними провайдерами туристичних послуг для збору всієї інформації від постачальників туристичних послуг, агрегує її і готує пропозицію поїздки.
7. Якщо взаємодія з провайдерами туристичних послуг пройшла успішно, VTA забезпечує агреговані пропозиції для клієнта. В іншому випадку, вибираються інші комбінації постачальників туристичних послуг (етап 4).
8. Клієнт отримує пропозицію для своєї поїздки (або повідомлення відмови про те, що немає пропозиції).

У цій роботі ми орієнтуємося на кроки 4 і 5 сценарію, описаного вище. Тобто, враховуючи мету, яка кодує запит користувача, ми покажемо, як можна вибрати набір відповідних туристичних сервісів і композувати їх для того, щоб генерувати виконуваний код, який виконує композицію цих сервісів відповідно до мети.

Ми припускаємо, що в домені подорожей використовується термінологія, яка базується на онтології, яка визначає основні поняття домену (екскурсії, проживання, клієнти ...). Ця онтологія описана надалі з використанням синтаксису WSML. Детальніше про моделювання онтології в WSML знаводиться в [LKMR +05]. Онтологія визначає концепти Client, Location, Trip (поїздки) і Accommodation (проживання), разом з їх атрибутами. Наприклад, Trip визначається унікальним ідентифікатором (це може бути політ або номер поїзда), датою і початком та метою розміщення. Концепт поїздки, яка була доступна, формалізується як концепт TripAvailable, який визначається як суб-концепт Trip. Аналогічно для концептів поїздки, які були заброньовані, але в даному випадку клієнт, який замовив поїздку стає атрибутом TripBooked. Подібні концепти вводяться також для проживання.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml−syntax/wsml−flight"
namespace {trv_"http://www.example.org/Travel"}
[...]
ontology trv#simpleTravelOntology
/* Client doing the travel */
concept trv#Client
   trv#name ofType _string
   trv#gender ofType _string
/* Destination of the travel */
concept trv#Location
   trv#name ofType _string
/* Trips */
concept trv#Trip
   trv#id ofType (1 1) _string
   trv#date ofType _date
   trv#start ofType trv#Location
   trv#destination ofType trv#Location
/* Accommodation */
concept trv#Accommodation
   trv#id ofType (1 1) _string
   trv#date ofType _date
   trv#location ofType trv#Location
/* Trips /accommodations being available */
concept trv#TripAvailable subConceptOf trv#Trip
concept trv#AccommodationAvailable subConceptOf trv#Accommodation
/* Trips /accommodations being booked */
concept trv#TripBooked subConceptOf trv#Trip
    trv#pax ofType (1 *) trv#Client
```

```
concept trv#AccommodationBooked subConceptOf trv#Accommodation
    trv#pax ofType (1 *) trv#Client
/* The first date is the user requested date, the second one is the trip date, the relation tells us if the two dates are
compatible (the trip date should contain the user requested date, but some additional days can be added before
and/or after the requested dates, e.g., due to constraints in the flights ) */

relation trv#Compatible(ofType _date, ofType _date)
```

Лістинг 1 – Основна онтологія подорожі

3.2 Моделювання Веб-сервісів

У наступних підрозділах ми докладно опишемо поведінкові аспекти різних Веб-сервісів, що реалізують деякі провайдери електронних послуг, з тим щоб дозволити композицію таких сервісів. Точніше, ми припускаємо, що існує три таких доступних Веб-сервіси: простий сервіс Flight Booking (Бронювання польоту), сервіс Train Booking (Бронювання залізничних квитків) та сервіс Hotel Booking (Бронювання готелю), які ми опишемо двома можливими способами. По-перше, ми опишемо, як такі послуги можна моделювати в машинній моделі на основі онтології, яка описує динамічний інтерфейс з Веб-сервісом в термінах онтології доступного абстрактного кінцевого автомату. Далі ми опишемо ту ж модель, використовуючи один з найбільш часто використовуваних стандартів у технологіях Веб-сервісів, а саме BPEL4WS.

У наступному розділі 4 ми покажемо, що обидві ці моделі дозволяють скорочення до кінцевих автоматів, на вершині яких можуть застосовуватися ефективні методи композиції рівня процесу.

3.2.1 Підхід WSMO

Поведінковий опис Веб-сервісу в WSMO ділиться на опис функціонального рівня, тобто *здатність* (capability) на високому рівні, і опис інтерфейсу, який моделює інтерфейс хореографії відповідного сервісу, тобто, як взаємодіяти з сервісом за допомогою станів для досягнення бажаних функціональних можливостей.

Введемо тепер концепти визначення повідомлень, які використовують окремі сервіси для взаємодії. Вони визначаються за допомогою онтологій, описаних в WSML. Ми описуємо онтологію для сервісу Flight - онтології для Train та Hotel є дуже схожими.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml−syntax/wsml−flight"
namespace {trv _"http://www.example.org/Travel",
fl _"http:// www.example.org/BookFlight"}
[...]
ontology fl#simpleFlightOntology
importsOntology trv#simpleTravelOntology
concept fl#Flight subConceptOf trv#Trip
        fl#flightNumber ofType (1 1) _string
axiom definedBy
        ?x[fl#flightNumber hasValue ?fn] memberOf fl#Flight implies ?x[trv#id hasValue ?fn] memberOf trv#Trip.
concept fl#FlightAvailable subConceptOf {trv#TripAvailable, fl#Flight}
        fl#seatNumber ofType _string
concept fl#FlightBooked subConceptOf {trv#TripBooked, fl#Flight}
        fl#seatNumber ofType _string
// The following concepts define the messages received/sent by the Flight service
concept fl#FlightRequest subConceptOf trv#Trip
        fl#client ofType trv#Client
concept fl#FlightNotAvailable subConceptOf trv#Trip
concept fl#FlightOffer subConceptOf {trv#Flight,fl#FlightRequest}
concept fl#FlightConfirm subConceptOf fl#FlightOffer
concept fl#FlightCancel subConceptOf fl#FlightOffer
```

Лістинг 2 – Онтологія Flight

Онтологія для Flight складається з двох частин. У першій частині основні концепти, введені для загальних поїздок в лістингу 1, уточнюються до конкретного випадку польотів. Унікальний ідентифікатор поїздки також визначається як номер рейсу. У другій частині вводяться додаткові концепти, які будуть використовуватися надалі у вигляді повідомлень у взаємодіях сервісу Flight з його інвокером.

WSMO описи можливостей

У WSMO можливість (capability) Веб-сервісу описується деякими нефункціональними властивостями і за допомогою умов, які повинні витримуватись перед тим, як сервіс може бути виконаний, і за допомогою результату, який досягається при виконанні сервісу. Опускаючи нефункціональні аспекти, які знаходяться поза сферою цієї роботи, WSMO ділить ці умови і результати на чотири категорії: *передумови* (*preconditions*) для визначення умов на інформаційному просторі (тобто, інформація, яка використовується для розрахунку), які повинні утримуватись до виконання, тобто вхід на вимогу сервісу; *припущення* (*assumptions*) як умови на світ (позначаючи аспекти, які не пов'язані з обчисленням), що доведеться тримати перед виконанням; *постумови* (*postconditions*) визначають умови на інформаційному просторі після виконання, тобто вихід сервісу, і *ефекти* (*effects*) як умови на світ, які тримаються після виконання сервісу. Ці важливі елементи з описів можливостей виражені у вигляді аксіом онтологій. Набір *розділених* (*shared*) *змінних* може бути оголошений, які є неявно квантифіковані і чия сфера є вся можливість Веб-сервісу. Неформально, логічна інтерпретація можливості Веб-сервісу є те, що при будь-яких значеннях, прийнятих розділеними змінними, передумови і припущення імплікують постумову і ефект.

Як приклад ми моделюємо тут можливість сервісу Flight Booking – знову Train та Hotel можуть бути змодельовані аналогічно.

```
namespace {trv _"http://www.example.org/Travel",
fl _"http://www.example.org/BookFlight"}
webService fl#BookFlight
capability fl#BookFlightCap
        sharedVariables {?req, ?date, ?start, ?dest, ?client}
        precondition definedBy
            ?req[
                trv#date hasValue ?date,
                trv#start hasValue ?start,
                trv#destination hasValue ?dest,
                fl#client hasValue ?client]
             memberOf fl#FlightRequest.
        assumption definedBy
            exists {?flight} ( ?flight [ trv#start hasValue ?start,
                        trv#destination hasValue ?dest,
                        trv#date hasValue ?date] memberOf fl#FlightAvailable).
        postcondition definedBy
                fl#offer (?req)[trv#date hasValue ?date,
                trv#start hasValue ?start,
                trv#destination hasValue ?dest,
                trv#flightNumber hasValue fl#fn(?req),
                fl#client hasValue ?client] memberOf fl#FlightOffer.
        effect definedBy
             fl#booking(?req)[ trv#date hasValue ?date,
                trv#start hasValue ?start,
                trv#destination hasValue ?dest,
                fl#flightNumber hasValue fl#fn(?req),
                trv#pax hasValue ?client] memberOf fl#FlightBooked.
```

Лістинг 3 – Сервіс Flight Booking: рівень можливості

Зверніть увагу на різні ролі передумови, припущення, постумови і ефекту. Передумова використовується для вираження обмежень на входи, які запитувач повинен бути в змозі

забезпечити для сервісу. Припущення виражає обмеження для успішного виконання сервісу, які запитувач не може контролювати (той факт, що політ насправді є доступним). Постумова висловлює нову інформацію, представлену запитувачу після успішного виконання сервісу (пропозиція з відповідною інформацією для запитувача, така як номер рейсу). Ефект, нарешті, висловлює реальні світові ефекти виконання Веб-сервісів, як той факт, що існує реальне замовлення для запитаного польоту.

Функціональна залежність між входом (запит) та параметрами вихід/ефекти моделюється функціональними символами в цьому прикладі, наприклад, символ функції **booking/1** залежить від ідентифікатора початкового запиту **?req**). Вище зазначена можливість говорить, що для будь-якого запиту із зазначеною датою, початок і кінець розташування, і клієнтом, пропозиція польоту буде надана як вихід і замовлення буде зроблене в припущенні, що відповідний політ доступний.

Зауважимо, що, на жаль, терміни, які використовуються для позначення чотирьох елементів, що використовуються для визначення можливостей Веб-сервісів в WSMO, відрізняються від «стандартних» термінів, які обговорювались в розділі 2.1. Передумови, зокрема, позначають різні елементи. WSMO передумови відповідають входам, WSMO припущення - передумовам, WSMO постумови - виходам, і WSMO ефекти - ефектам.

WSMO описи інтерфейсу хореографії (WSMO choreography interface descriptions)

На рівні можливостей, неможливий дрібно структурний опис складних шаблонів взаємодії з сервісом. Можливість тільки дає крупнозернистий опис оголошеної функціональної можливості сервісу, опускаючи кроки взаємодії, які повинні бути прийняті з метою досягнення цієї функціональності або можливості відмови, тобто опис рівня можливості охоплює тільки послідовності «успішної» взаємодії в плані того, що сервіс хоче рекламувати.

Проте, для того, щоб з'ясувати, як насправді виконати можливо складний сервіс, потрібно більш дрібнозернистий опис. Цей опис процесу взаємодії із станом (stateful interaction process) з сервісом може бути виражений, наприклад, у workflow подібній мові, як BPEL4WS. WSMO використовує інший підхід, де описані такі взаємодії з точки зору абстрактних машин станів (abstract state machines) [BS03].

Надалі ми коротко резюмуємо модель опису хореографії WSMO. Для більш детальної інформації про модель, яка використовується тут, ми звернемося до поточного проекту робочої групи WSMO [FSPR05]. Синтаксис, який використовується в даному проекті (а також у цій роботі), все ще може бути предметом незначних змін, але досить стабільний, щоб проілюструвати використовувану модель. Ми вкажемо, коли ми відхилимось від запропонованого синтаксису в [FSPR05], і пояснимо в деталях відповідні розширення, які ми вважаємо необхідними.

Основні абстрактні машини станів (Basic Abstract State Machines). Абстрактні машини станів (ASMs для стислості), раніше відомі як алгебри еволюції (Evolving Algebras) [BS03], забезпечують засоби для опису систем в точний спосіб з використанням семантично обґрунтованої математичної нотації. Основні принципи полягають у визначенні заземлених моделей і проектування систем шляхом уточнень. Заземлені моделі визначають вимоги й операції системи, які виражені в математичній формі. Уточнення дозволяють вираз класичної методології розділяй і володарюй (classical divide and conquer methodology) для проектування системи в точній нотації, яка може бути використана для абстракції, валідації та верифікації системи на даному етапі в процесі розробки.

Абстрактні машини станів діляться на дві основні категорії, а саме основні ASMs і мультиагентні ASMs. Розробник виражає поведінку системи в середовищі. Багатоагентні ASMs дозволяють виразити поведінку системи в термінах кількох сутностей, які співпрацюють для досягнення функціональності. Для опису поведінки однієї сторони, ми просто зацікавлені в основних ASMs.

Основна ASM визначається в термінах сигнатури стану плюс кінцевий набір правил переходу, які виконуються паралельно. Вона може включати недетерміновану поведінку. Кінцеві автомати (finite state machines) можна розглядати як окремий випадок таких основних ASMs.

Сигнатура стану для класичних ASMs зазвичай просто складається з набору статичних і динамічних функцій, які можуть бути локально оновлені за допомогою так званих правил оновлення. Динамічні функції поділяються на чотири інші категорії, а саме, *контрольовані* (*controlled*), *моніторені* (*monitored*) (або *в* (*in*)), *взаємодія* (*interaction*) (або *загальний* (*shared*)) і *зовнішній* (*out*). Контрольовані функції є безпосередньо обновлюваними тільки за правилами машини *M*. Таким чином, вони не можуть бути ні читані, ні актуалізовані середовищем. Моніторені функції можуть бути оновлені тільки навколишнім середовищем і прочитані машиною *M* і, отже, являють собою зовнішньо контрольовану частину стану. Загальні функції можуть бути прочитані і оновлюються як навколишнім середовищем, так і правилами машини *M*. Зовнішні функції можуть бути оновлені, але не читаються *M*, але можуть бути прочитані навколишнім середовищем. Крім того, ASMs визначають так звані похідні (derived) функції. Ці функції не оновлювані машиною чи навколишнім середовищем, але які визначені в термінах інших статичних та динамічних (і похідних) функцій.

Більшість основних правил є Оновлення (Updates), які приймають форму призначень (assignment) (так званих оновлень функції) наступним чином:

$f(t_1, ..., t_n) := v$,

де f є *n*-арна функція (вказана сигнатурою) і терми $t_1, ..., t_n$ позначають *місце* (*location*), де динамічна f повинна бути оновлена до нового значення v, визначаючи значення функції $f(t_1, ..., t_n)$ у наступному стані. Пари *місце-значення* (location-value) $((t_1, ..., t_n), v)$ називаються основними оновленнями і являють собою основну одиницю зміни стану в ASM.

Більш складні правила переходу визначаються таким чином рекурсивно. (Зауважимо, що заради ясності, ми злегка відхилилися тут від початкового синтаксису, який використовується в [BS03]). По-перше, правила переходу можуть бути захищені *Condition* наступним чином:

if *Condition* then *Rules* endIf.

Тут *Condition* довільна замкнута формула першого порядку (або будь-яка інша логіка, що лежить в основі сигнатури), тобто концепт ASMs відділяється від основного логічного формалізму. Таке захищене правило переходу має семантику, що *Rules* в своїй області виконуються паралельно, кожен раз, коли *Condition* виконується в поточному стані.

Далі, основні ASMs дозволяють деяку форму універсально квантифікованого паралелізму за допомогою правил переходу виду

forAll *Variable* with *Condition* do *Rules* endForall

Тут *Variable* є змінна, що входить вільно в *Condition*, зі змістом, що *Rules*[*Variable*/*Value*] виконуються паралельно для всіх можливих зв'язків *Variable* до конкретного *Value* таким чином, що *Condition* [*Variable*/*Value*] виконується в поточному стані. Тут *Condition* [*Variable*/*Value*] (і *Rules*[*Variable*/*Value*], відповідно) стоїть для умови (або правила, відповідно), де кожне входження *Variable* замінюється *Value*.

Точно так само, основні ASMs дозволяють недетермінований вибір за допомогою правил переходу форми

choose *V ariable* with *Condition* do *Rules* endChoose

Тут, на відміну від правила **forAll**, одне з можливих зв'язувань *V ariable* таким чином, що *Condition* тримається, визначено недетерміновано машиною, і *Rules* виконуються паралельно тільки для цього окремого зв'язку.

Один крок виконання ASM можна резюмувати наступним чином:

1. Розгортаються правила, відповідно з поточним станом та умовами, що виконуються в цьому стані, до набору основних оновлень.

2. Виконати одночасно всі оновлення.

3. Якщо оновлення узгоджуються (тобто, ніякі два різних оновлення не оновлюють те ж саме місце з різними значеннями, що означає, що не повинно бути пари оновлень (loc, v), (loc, v') з $v \neq v'$), то результат виконання дає наступний стан.

4. Всі місця, на які не діяли оновлення, зберігають свої значення.

Ці кроки повторюються, поки жодна з умов будь-якого правила не оцінюється як істина, тобто розгортання призводить до порожнього набору оновлень. У разі суперечливих оновлень, виконання машини є недійсним.

Читаність і структура загальних ASMs може бути поліпшена шляхом введення так званих контрольних станів як синтаксичного цукру (syntactic sugar). Такі контрольні стани дозволяють розглядати ASMs як прості розширення кінцевих автоматів і, таким чином, мають бажані властивості як високорівневе графічне представлення і модульність машини. Control State ASM є ASM з однією конкретною контрольованою функцією *ctlState* (яка має в якості свого діапазону кінцеву кількість цілих чисел або кінцеве перерахування стан-дескриптори (state-descriptors)) і кожне правило переходу має форму:

```
if ctlState = i then
        if cond1 then
                rule1
                ctlState := j1
        endIf
        ...
        if condn then
                rulen
                ctlState := jn
        endIf
endIf
```

де i, j_1, ..., j_n є дескрипторами контрольних станів і $rule_1$, ..., $rule_n$ є набори правил.

В принципі, ASMs контрольного стану є FSMs, збагачені синхронним паралелізмом і обробкою даних (і, таким чином, можливо, знову нескінченно). Зверніть увагу, що ASMs контрольного стану не є виразним обмеженням загальних ASMs, але роблять його легшим, щоб визначити потік управління.

WSMO модель хореографії (**WSMO Choreography Model**). Хореографічна частина інтерфейсу сервісу описує поведінку сервісу з точки зору клієнта, це визначення є відповідним до наступного визначення, даного W3C Glossary (http://www.w3.org/TR/ws-gloss/): *Хореографія Веб-сервісів стосується взаємодії сервісів з їх користувачами. Будь-який користувач Веб-сервісу, автоматизований або інший, є клієнтом цього сервісу. Ці користувачі можуть, у свою чергу, бути й іншими Веб-сервісами, застосуваннями або людськими істотами.*

З цією метою, на основі базової моделі ASM, інтерфейс хореографії в WSML описується як ASM, який діє на концептах і відношеннях WSML онтології, щоб описати сигнатуру стану. За аналогією з класифікацією функції в ASMS, концепти і відношення в WSML інтерфейсах хореографії класифікуються як *in*, *out*, *controlled*, *shared*, *static*.

Стан для даної сигнатури WSMO хореографії визначається всіма допустимими ідентифікаторами WSMO, концептами, відношеннями і аксіомами. Елементи, які можна змінити, і, які використовуються, щоб виразити різні стани хореографії, є екземплярами концептів і відношень, які використовуються аналогічним чином в місцях (locations) в ASMs. Ці зміни виражаються в термінах створення нових екземплярів або зміни значень атрибутів.

Для того, щоб зв'язати з традиційними описами інтерфейсу Веб-сервісу, таких як WSDL, WSMO дозволяє зв'язати концепти класів *in*, *out* і *shared* , які прив'язані до входів і виходів операцій в опису WSDL відповідного сервісу.

На відміну від основних ASMs, сама основна форма правил не є призначеннями, але ми маємо справу з основними операціями на даних екземпляра в онтологіях, таких як додавання, видалення і оновлення екземплярів у онтологію сигнатури. З цією метою ми визначаємо атомарні функції поновлення для додавання, видалення і оновлення екземплярів, які дозволяють нам робити додавання і видалення екземплярів в/з концептів і відношень, і додавати і видаляти значення

атрибутів для окремих екземплярів. У WSMO хореографії ці основні оновлення визначаються як набір модифікаторів фактів, які бувають трьох різних типів.

1. add(*fact*)
2. delete(*fact*)
3. update(*fact*(*old* => *new*)), or simply
4. update(*fact*)

fact може бути або фактом членства (x memberOf y або r (t_1, ..., t_n) для n-арного відношення r, відповідно), фактом атрибуту (x [a hasValue y]) для концептів, або комбінацією фактів членства концептів і значень атрибутів у вигляді молекули WSML, скорочуючи кон'юнкції фактів членства і атрибутів, див. [LKMR +05]. Більш складні правила переходу визначаються рекурсивно, аналогічно класичним ASMs за допомогою правил **if-then**, **forAll-do** і **choose-do**. Зверніть увагу, що нинішнє визначення в останній версії з [FSPR05] дозволяє тільки add і delete, але ми додаємо примітив оновлення для зручності моделювання.

Правило update форми 3 може приймати одну з наступних форм:

update(r(t_1, ..., t_{i-1},$t_{i\ old}$ => $t_{i\ new}$, t_{i+1}, ..., t_n)), яка є скороченням для набору правил
 delete(r(t_1, ..., t_{i-1},$t_{i\ old}$, t_{i+1}, ..., t_n))
 add(r(t_1, ..., t_{i-1},t_{inew} , t_{i+1}, ..., t_n))
update(x memberOf y_{old} => y_{new}))), яка є скороченням для набору правил
 delete(x memberOf y_{old})
 add(x memberOf y_{new})
update(x[a hasValue y_{old} => y_{new}]), яка є скороченням для набору правил
 delete(x[a hasValue y_{old}])
 add(x[a hasValue y_{new}])

Правило update форми 4 може приймати одну з наступних форм:

update(r(t_1, ..., t_n)), яка є скороченням для правил
 forall {?x1, ... ,?xn} with ?x1 $\neq$ t_1 or ... or ?xn $\neq$ t_n do delete(r(?x1, ...,?xn))
 endForall
 add(r(t_1,. . . ,t_n))
update(x memberOf y), яка є скороченням для правил
 forall ?y1 with ?y1 $\neq$ y do delete(x memberOf ?y1) endForall
 add(x memberOf y)
update(x[a hasValue y]), яка є скороченням для правил
 forall ?y1 with ?y1 $\neq$ y do delete(x[a hasValue ?y1]) endForall
 add(x[a hasValue y])

У порівнянні з основними ASMs, в WSMO хореографії діють такі обмеження до умов і змінних.

(WSML повна) умова ((WSML Full) Condition) є обмежена форма WSML логічних виразів, де всі вільні змінні, які не пов'язані, будуть огороджувальним вибором або forAll конструкції інтерпретуються як екзистенційно кваліфіковані.

WSML умова ядра (WSML Core Condition) є WSML повний логічний вираз, який складається тільки з молекул, побудованих з атомів **memberOf** і **hasValue** та логічних зв'язок **and** і **or**, де всі незв'язані змінні є екзистенційно квантифіковані (тобто, умова є кон'юнктивний запит).

Для подальшої зручності моделювання введемо наступне скорочення для моделювання загального безумовного недетермінізму, який часто використовується в кінцевих автоматах і особливо корисно для описів інтерфейсу:

$$rule_1|rule_2| \dots |rule_n$$

є скорочення для

 choose ?x with ?x = 1 or ?x = 2 or ?x = n do
 if ?x= 1 then *rule*$_1$ endIf

```
if ?x= 2 then rule1 endIf
...
if ?x= n then rulen endIf
endChoose
```

Семантика WSMO ASMs визначається аналогічно до основних ASMs.

Стан S для даної сигнатури WSMO хореографії визначається всіма допустимими WSMO ідентифікаторами, концептами, відношеннями і аксіомами. Елементи, які можна змінити, і, що використовуються, щоб виразити різні стани хореографії, є екземплярами концептів і відношень, які використовуються аналогічно в місцях (locations) в ASMs. Ці зміни виражаються в термінах створення нових екземплярів або змін значень атрибутів.

Виконання інтерфейсу хореографії визначається аналогічно виконанням основних ASMs в [BS03], тобто виконання визначені як послідовності можливих кроків одноразового виконання.

Можливі кроки виконання визначаються наступним чином:

$$S' = S \setminus \{fact | \text{delete}(fact) \in U\} \cup \{fact | \text{add}(fact) \in U\}$$

де S - поточний стан, U - є узгоджений набір оновлення, S' - є результуючий стан застосування U в S.

Нагадаємо, що інтерфейс хореографії визначається набором правил переходу R: нехай O позначає імпортовану онтологію (-ї) сигнатури і S позначає поточний стан. Визначимо набори оновлення для (R, S) індуктивно:

$$U(\text{add}(fact), S) = \text{add}(fact)$$
$$U(\text{delete}(fact), S) = \text{delete}(fact)$$
$$U(R, S) = \cup_{r \in R} U(r, S)$$

$$U(\text{if } Cond \text{ then } R, S) = \begin{cases} U(R,S) \text{ if } O \cup S \models Cond \\ \emptyset \text{ otherwise} \end{cases}$$

$$U(\text{forall } ?Var \text{ with } Cond \text{ do } R \text{ endForall}, S) = \{U_{R\theta,S} | \theta \text{ such that } \theta = \{?Var/id\} \text{ where } id \text{ is an identifier such that } O \cup S \models Cond\theta\}$$

$$U(\text{choose } ?Var \text{ with } Cond \text{ do } R \text{ endChoose}, S) = \begin{cases} U(R\theta, S) \text{ where } \theta = \{?Var/id\} \text{ with } id \text{ being a} \\ \text{non-deterministically chosen identifier such that } O \cup S \models Cond \\ \emptyset \text{ if } O \cup Scup\{exists?Var(Cond)\} \text{ is unsatisfiable.} \end{cases}$$

Оновлений набір U є сумісним, якщо він не містить які-небудь два елементи add(*fact*) і delete(*fact*), і результуючий стан

$$S' = S \setminus \{fact | \text{delete}(fact) \in U\} \cup \{fact | \text{add}(fact) \in U\}$$

є сумісним з онтологією сигнатури, тобто, $S' \cup O$ є виконуваним.

Перейдемо до опису інтерфейсу хореографії сервісу Flight Booking. Для опису цього інтерфейсу ми використовуємо синтаксис, визначений вище, з іншим невеликим розширенням до [FSPR05] ASMs контрольного стану, використовуючи нове ключове слово ctl_state, щоб перерахувати можливі дескриптори контрольного стану.

```
interface fl#BookFlightInterface
    choreography
        stateSignature
            importsOntology fl#simpleFlightOntology
        in
            fl#FlightRequest withGrounding ”http ://... ” ,
            fl#FlightConfirm withGrounding ”http ://... ” ,
            fl#FlightCancel withGrounding ” http ://... ”
        out
            fl#FlightNotAvailable withGrounding ”http ://... ” ,
            fl#FlightOffer withGrounding ”http ://... ”
        shared
            fl#Flight ,
            fl#FlightAvailable ,
            fl#FlightBooked
        ctl_state {fl#start ,fl#offerMade,fl#noAvail ,fl#confirmed,fl#cancelled}
        transitionRules
```

```
            if ( ctl_state = fl#start ) then
              forall {?req,?date,?start,?dest,?client} with
                   ?req[trv#date hasValue ?date,
                   trv#start hasValue ?start,
                   trv#destination hasValue ?dest,
                   fl#client hasValue ?client] memberOf fl#FlightRequest
              do
                   if exists {?f} (?f [trv#date hasValue ?date,
                             trv#start hasValue ?start,
                             trv#destination hasValue ?dest] memberOf
                             fl#FlightAvailable ) then
                     choose {?fn} with
                             exists {?f} (?f [trv#date hasValue ?date,
                                     trv#start hasValue ?start,
                                     trv#destination hasValue ?dest,
                                     fl#flightNumber hasValue ?fn] memberOf
                                     fl#FlightAvailable )
          do
            add( fl#offer (?req)[trv#date hasValue ?date,
                   trv#start hasValue ?start,
                   trv#destination hasValue ?dest,
                   fl#flightNumber hasValue ?fn,
                   fl#client hasValue ?client] memberOf
                   fl#FlightOffer )
            ctl_state := fl#offerMade
                   endChoose
          else
                   add(fl#notAvailable(?req)[trv#date hasValue ?date,
                             trv#start hasValue ?start,
                             trv#destination hasValue ?dest] memberOf
                             fl#FlightNotAvailable )
                   ctl_state := fl#noAvail
          endIf
      endForall
    endIf
  if ( ctl_state = fl#offerMade) then
    forall {?offer , ?client} with ( ?offer [ fl#client hasValue ?client]
          memberOf {fl#FlightConfirm,fl#FlightOffer}) do
          add(?offer[trv#pax hasValue ?client] memberOf trv#FlightBooked)
          ctl_state := fl#confirmed
    endForall
  endIf
  if ( ctl_state = fl#offerMade) then
    forall {?offer} with ( ?offer memberOf {fl#FlightCancel,fl#FlightOffer}) do
          ctl_state := fl#cancelled
    endForall
  endIf
```

Лістинг 4 – Сервіс Flight Booking: рівень інтерфейсу

Розгалуження на умові

```
if exists {?f} (?f[trv#date hasValue ?date,
        trv#start hasValue ?start,
        trv#destination hasValue ?dest] memberOf
        fl#FlightAvailable) then
[. . . ]
else
[. .
endIf
```

в контрольному стані fl#start в наведеному вище лістингу є повністю недетермінованим для користувача сервісу, оскільки воно залежить від загального концепту доступних рейсів, який є фактично відомим тільки постачальнику сервісу. Таким чином, ми могли в рівній мірі переформулювати цю частину опису інтерфейсу хореографії за допомогою недетермінірованного оператору вибору, визначеного вище:

```
if ( ctl_state = fl#start ) then
  forall {?req,?date,?start,?dest,?client} with
        ?req[trv#date hasValue ?date,
        trv#start hasValue ?start,
        trv#destination hasValue ?dest,
        fl#client hasValue ?client] memberOf fl#FlightRequest
  do
     add( fl#offer (?req)[trv#date hasValue ?date,
              trv#start hasValue ?start,
```

```
                        trv#destination hasValue ?dest,
                        fl#flightNumber hasValue fl#fn(?req),
                        fl#client hasValue ?client] memberOf fl#FlightOffer)
        ctl_state := fl#offerMade
|
        add(fl#notAvailable(?req)[trv#date hasValue ?date,
                        trv#start hasValue ?start,
                        trv#destination hasValue ?dest]
            memberOf fl#FlightNotAvailable)
        ctl_state := fl#noAvail
enfIf
```

Лістинг 5 – Сервіс Hotel Booking – безумовно недетермінована версія

Цей коротший опис містить менш семантичну інформацію. Більше того, він не дозволяє використовувати *припущення*, зазначене в можливості в лістингу 3, тобто, що ми можемо очікувати, що рейси, як правило, доступні. Подальша розробка зв'язку між можливістю та описом інтерфейсу знаходиться на нашому порядку денному.

Ми опускаємо опис можливості і рівня інтерфейсу для сервісу Train Booking, який ми приймаємо, що він схожий до Flight. Ми повідомляємо замість цього опис сервісу Hotel Booking.

```
namespace {trv _"http://www.example.org/Travel",
            htl _"http :// www.example.org/BookHotel"}
webService htl#BookHotel
capability htl#BookHotelCap
            sharedVariables {?req,?date, ?loc, ?client}
            precondition definedBy
               ?req[trv#date hasValue ?date,
               trv#location hasValue ?loc,
               htl#client hasValue ?client]
            memberOf htl#HotelRequest.
            assumption definedBy
               exists {?hotel} (?hotel[trv#date hasValue ?date,
                        trv#location hasValue ?loc]
                        memberOf htl#HotelAvailable).
            postcondition definedBy
               htl#offer (?req)[trv#date hasValue ?date,
               trv#location hasValue ?loc,
               htl#client hasValue ?client]
              memberOf htl#HotelOffer.
            effect definedBy
                htl#booking(?req)[trv#date hasValue ?date,
                        trv#location hasValue ?loc,
                        htl#hotelName hasValue htl#hn(?req),
                        trv#pax hasValue ?client]
            memberOf trv#HotelBooked.

            interface htl#BookHotelInterface

choreography
            stateSignature
                importsOntology htl#simpleHotelOntology
                in
                   htl#HotelRequest withGrounding "http ://... " ,
                        htl#HotelConfirm withGrounding "http ://... " ,
                        htl#HotelCancel withGrounding " http ://... "
                 out
                        htl#HotelNotAvailable withGrounding "http ://... " ,
                        htl#HotelOffer withGrounding "http ://... "
                 shared
                        htl#Hotel ,
                        htl#HotelAvailable ,
                        htl#HotelBooked
                  ctl_state {htl#start ,htl#offerMade,htl#noAvail ,htl#confirmed,htl#cancelled}

                  transitionRules
                        if ( ctl_state = htl#start ) then
                             forall {?req,?date,?loc,?client} with
                                     ?req[trv#date hasValue ?date,
                                     trv#location hasValue ?loc,
                                     htl#client hasValue ?client] memberOf htl#HotelRequest
                             do
                                     add(htl#offer (?req)[trv#date hasValue ?date,
                                                 trv#hotelName hasValue ?name,
                                                 trv#location hasValue ?loc,
                                                 htl#client hasValue ?client] memberOf htl#HotelOffer)
                                     ctl_state := htl#offerMade
```

```
                        |
                        add(htl#notAvailable(?req)[trv#date hasValue ?date,
                                trv#location hasValue ?loc] memberOf htl#HotelNotAvailable)
                        ctl_state := htl#noAvail
                endForall
        endIf
if ( ctl_state = htl#offerMade) then
        forall {?client , ?offer} with ( ?offer [ htl#client hasValue ?client] memberOf {htl#HotelConfirm,
                        htl#HotelOffer})
        do
                        add(?offer[trv#pax hasValue ?client] memberOf trv#HotelBooked)
                        ctl_state := htl#confirmed
        endForall
endIf
if ( ctl_state = htl#offerMade) then
forall {?offer} with ?offer memberOf {htl#HotelCancel,htl#HotelOffer} do
ctl_state := htl#cancelled
endForall
endIf
```

Лістинг 6 – Сервіс Hotel Booking

3.2.2 Підхід BPEL4WS

BPEL4WS [ACD +03] забезпечує операційний опис (stateful – перевірка стану) поведінки Веб-сервісів на вершині інтерфейсів сервісів, визначених у своїх WSDL специфікаціях. BPEL4WS опис сервісу визначає партнерів сервісу, його внутрішні змінні та операції, які спрацьовують при виклику сервісу деякими партнерами. Операції включають присвоєння змінних, виклики інших сервісів і отримання відповіді, породження паралельних потоків виконання, і недетермінованого вибору одного серед різних напрямків дій. Стандартні імперативні конструкції, такі як, if-then-else, case choices і loops (петлі) також підтримуються.

У лістингу 7 ми повідомляємо WSDL специфікацію сервісу Flight Booking[1]. Файл починається з опису структурних повідомлень, відповідних взаємодії з сервісом. Далі, WSDL специфікація визначає операції виклику і відповіді, які забезпечуються сервісом. Операції зібрані в типи портів (port types), які пов'язані з різними каналами зв'язку сервісу бронювання польоту із своїми партнерами. У нашому прикладі ми визначаємо два типи портів, а саме Flight_PT для вхідних запитів і повідомлень і Flight_CallbackPT для вихідних повідомлень від польоту до сервісу, що викликає. Нарешті, WSDL специфікація визначає двонаправлені посилання (links) між сервісом з бронювання авіаквитків і його партнерами. У нашому випадку є тільки одне з таких посилань, між сервісом бронювання польоту і замовником, який викликає його.

Лістинг 8 надає BPEL4WS специфікацію, що, спираючись на верхню частину WSDL специфікації, описує протокол, якого потрібно дотримуватися, щоб взаємодіяти із сервісом замовленням польоту. Специфікація BPEL4WS починається з оголошення партнерів, які беруть участь у взаємодії, і посилань та ролей серед них (тут використовуються WSDL описи partnerLinkType): у нашому випадку, єдиним партнером є клієнт, який викликає сервіс. Тоді визначається набір змінних. Ці змінні використовуються у взаємодіях, як контейнери для вхідних і вихідних повідомлень, а також для внутрішніх обчислень сервісу.

Основна частина BPEL4WS специфікації містить специфікацію послідовності (або потоку) дій, які визначають взаємодії з сервісом. Цей опис надається з точки зору сервісу бронювання авіаквитків. Виконання сервісу починається з отримання повідомлення запиту від клієнта. Сервіс бронювання авіаквитків вирішує всередині, чи є доступні рейси (switch перемикач, поіменований checkAvailability). Зауважимо, що BPEL4WS код вказує, що це внутрішнє рішення приймається сервісом польоту, але він не виставляє «як» це рішення приймається. Дійсно, остання інформація не потрібна клієнтові для взаємодії з сервісом польоту.

У разі відсутності польоту, сервіс з бронювання авіаквитків відповідає клієнту з повідомленням flightNotAvailable і закінчується. В іншому випадку, визначається пропозиція і відправляється клієнтові. Зауважимо, що визначення пропозиції, яке здійснюється в дії assign, є непрозорим (opaque), тобто відомості про те, як пропозиція визначається, не виставляються.

[1] Для зручності читання ми спростили WSDL специфікацію і відповідний BPEL4WS код, залишивши без уваги деякі технічні частини.

Аналогічно до перевірки доступних рейсів, деталі про те, як пропозиція визначається, є внутрішніми до реалізації сервісу і не повинні бути розкриті клієнту.

BPEL4WS специфікація описує дуже детально взаємодії, які повинні бути здійснюватися з Веб-сервісом, щоб використовувати його. Тим не менш, це все ще не достатньо, щоб дозволити автоматичне композування такого Веб-сервісу з іншими сервісами. Дійсно, необхідно описати також «семантичні» аспекти таких взаємодій. Ми це робимо за рахунок розширення BPEL4WS специфікації «семантичними анотаціями» (див. також [PTBM05a]). Це необхідно, насамперед, щоб зв'язати «семантичний» сенс з вхідними і вихідними операціями, визначеними у файлі WSDL: це робиться в анотованій WSDL специфікації, наведеній в лістингу 9, який заснований на WSDL-S підході [SVMR05]. Можна помітити, що, в той час, як обидві частини start і client повідомлення запиту мають один і той же тип (вони є рядками), семантичні анотації визначають їх роль у визначенні поїздки (тобто, початок поїздки і клієнта поїздки).

```
<definitions name="Flight">
       <message name="flightRequestMsg">
          <!-- Flight booking request -->
          <part name="date" type="xsd:string"/>
           <part name="start" type="xsd:string"/>
           <part name="destination" type="xsd:string"/>
           <part name="client" type="xsd:string"/>
       </message>
       <message name="flightOfferMsg">
           <!-- Offer from the flight booking service -->
           <part name="flightNo" type="xsd:integer"/>
       </message>
       <message name="flightNotAvailableMsg">
            <!-- Answer message from the flight booking service if no flight is available -->
       </message>
       <message name="flightConfirmMsg">
             <!-- Flight booking confirmation from the requester -->
       </message>
       <message name="flightCancelMsg">
             <!-- Flight booking cancellation, if the requester is not interested in the offer received
       from the booking service -->
       </message>
       <portType name="Flight_PT">
              <operation name="flightRequest">
                  <input message="tns:flightRequestMsg"/>
              </operation>
              <operation name="flightConfirm">
                  <input message="tns:flightConfirmMsg" />
              </operation>
              <operation name="flightCancel">
                  <input message="tns:flightCancelMsg" />
              </operation>
       </portType>
       <portType name="Flight_CallbackPT">
              <operation name="flightOffer">
                   <input message="tns:flightOfferMsg" />
              </operation>
              <operation name="flightNotAvailable">
                   <input message="tns:flightNotAvailableMsg" />
              </operation>
       </portType>
       <plnk:partnerLinkType name="FlightRequest_PLT">
              <plnk:role name="FlightRequest_Server">
                 <plnk:portType name="tns:Flight_PT"/>
              </plnk:role>
              <plnk:role name="FlightRequest_Client">
                 <plnk:portType name="tns:Flight_CallbackPT"/>
              </plnk:role>
       </plnk:partnerLinkType>
</definitions>
```

Лістинг 7 – WSDL визначення сервісу *Flight*

```
<process name="Flight">
       <partnerLinks>
              <partnerLink name="client" partnerLinkType="FlightRequest PLT"
                     myRole="FlightRequest Server" partnerRole="FlightRequest Client"/>
       </partnerLinks>
<variables>
```

```
        <variable name="req" messageType="flightRequest"/>
        <variable name="offer" messageType="flightOffer"/>
</variables>
<sequence name="main">
        <receive operation="flightRequest" variable="req" partnerLink="client"/>
        <switch name="checkAvailability">
            <case name="isNotAvailable">
               <invoke operation="flightNotAvailable" partnerLink="client"/>
            </case>
            <otherwise name="isAvailable">
               <assign name="prepareOffer">
                    <copy><from opaque="yes"/>
                       <to variable="offer" part="fl"/></copy>
               </assign>
               <invoke operation="flightOffer" inputVariable="offer" partnerLink="client"/>
               <pick name="waitAcknowledge">
                   <onMessage operation="flightConfirm" partnerLink="client"/>
                   <onMessage operation="flightCancel" partnerLink="client"/>
               </pick>
           </otherwise>
        </switch>
    </sequence>
</process>
```

Лістинг 8 - BPEL процес сервісу *Flight*

```
<definitions name="Flight">
        <message name="flightRequestMsg">
                <!-- Flight booking request -->
                <part name="date" type="xsd:string" wssem:modelReference="trv#Trip[date]"/>
                <part name="start" type="xsd:string" wssem:modelReference="trv#Trip[start]"/>
                <part name="destination" type="xsd:string"
                         wssem:modelReference="trv#Trip[destination]"/>
                <part name="client" type="xsd:string" wssem:modelReference="trv#Client"/>
        </message>
        <message name="flightOfferMsg">
                <!-- Offer from the flight booking service -->
                 <part name="flightNo" type="xsd:integer" wssem:modelReference="trv#Trip[id]"/>
        </message>
        <message name="flightNotAvailableMsg">
                <!-- Answer message from the flight booking service if no flight is available -->
        </message>
        <message name="flightConfirmMsg">
                <!-- Flight booking confirmation from the requester -->
        </message>
        <message name="flightCancelMsg">
                <!-- Flight booking cancellation, if the requester is not interested in the offer received
                from the booking service -->
        </message>
        ....
</definitions>
```

Лістинг 9 – Анотований WSDL опис сервісу *Flight*

Друге використання семантичних анотацій полягає у визначенні результату взаємодії з Веб-сервісом. У нашому прикладі видно, що рейс був заброньований тільки, якщо рейс є доступним, сервіс польоту посилає пропозицію, і клієнт підтверджує прийняття пропозиції. Щоб виразити це в BPEL4WS специфікації, ми повинні диференціювати можливі кінцеві стани взаємодій з сервісом бронювання польоту. У лістингу 10 це виходить через семантичні анотації, які пов'язані з «dummy» (пустушка) empty дією, яка помічає собою успішний кінцевий стан взаємодії. Зверніть увагу на використання оператора add, вже описаного в попередньому підрозділі для підходу WSMO.

```
<process name="Flight">
        <partnerLinks>
             <partnerLink name="client" partnerLinkType="FlightRequest PLT"
               myRole="FlightRequest Server" partnerRole="FlightRequest Client"/>
        </partnerLinks>
        <variables>
                <variable name="req" messageType="flightRequest"/>
                <variable name="offer" messageType="flightOffer"/>
        </variables>
        <sequence name="main">
```

```
            <receive operation="flightRequest" variable="req" partnerLink="client"/>
            <switch name="checkAvailability">
                <case name="isNotAvailable">
                    <invoke operation="flightNotAvailable" partnerLink="client"/>
                </case>
                <otherwise name="isAvailable">
                   <assign name="prepareOffer">
                          <copy><from opaque="yes"/>
                          <to variable="offer" part="fl"/></copy>
                   </assign>
                   <invoke operation="flightOffer" inputVariable="offer" partnerLink="client"/>
                   <pick name="waitAcknowledge">
                      <onMessage operation="flightConfirm" partnerLink="client">
                          <empty wssem:effect="forall ?tr with (?tr memberOf trv#TripBooked
                             and ?tr[trv#date hasValue '/req/date',
                              trv#start hasValue '/req/start',
                               trv#destination hasValue '/req/destination',
                               trv#id hasValue '/offer/flightNo',
                               trv#client hasValue '/req/client']) add(?tr)"/>
                      </onMessage>
                     <onMessage operation="flightCancel" partnerLink="client"/>
                   </pick>
            </otherwise>
        </switch>
     </sequence>
</process>
```

Лістинг 10 - BPEL процес сервісу *Flight*

3.3 Моделювання мети композиції

У цьому розділі ми опишемо один з входів інтегрованої процедури виявлення і композиції, а саме зазначення мети та формалізація запиту замовника. Мета визначається VTA, відповідно до запиту замовника. Ми не обговорюємо тут в деталях, як виходить ця мета. Можливо також, що VTA має набір цілей (або шаблонів мети), які пов'язані з різними запитами подорожей, що клієнт може запропонувати. Формалізація мети запиту замовника визначається на вершині цієї онтології таким чином.

```
compositionGoal BookTrip
        sharedVariables {?date, ?start, ?dest, ?client}
        precondition definedBy
                ?date memberOf date and
                ?start memberOf trv#Location and
                ?dest memberOf trv#Location and
                ?client memberOf trv#Client
        assumption definedBy
                exists {?t, ?a, ?d} {
                        ?d memberOf date and trv#Compatible(?date,?d) and
                        ?t [ trv#start hasValue ?start,
                        trv#destination hasValue ?dest,
                        trv#date hasValue ?d] memberOf trv#TripAvailable and
                        ?a [ trv#location hasValue ?dest,
                        trv#date hasValue ?d] memberOf trv#AccommodationAvailable
        effect definedBy
                exists {?t, ?a, ?d} {
                        ?d memberOf date and trv#Compatible(?date,?d) and
                        ?t [ trv#start hasValue ?start,
                        trv#destination hasValue ?dest,
                        trv#date hasValue ?d,
                        trv#client hasValue ?client] memberOf trv#TripBooked and
                        ?a [ trv#location hasValue ?dest,
                        trv#date hasValue ?d,
                        trv#client hasValue ?client] memberOf trv#AccommodationBooked
        recovery definedBy
                (neg exists ?t [ trv#client hasValue ?client] memberOf trv#TripBooked) and
                (neg exists ?a [ trv#client hasValue ?client] memberOf trv#AccommodationBooked)
```

Лістинг 11 – Мета композиції

Змінні sharedVariables відповідають входам замовника і precondition визначає умови на цих змінних (наприклад, їх типи). Припущення assumption визначає умову, при якій композиція повинна успішно завершитися, повертаючи пропозицію замовнику - в нашому прикладі, якщо є підходящі поїздка та доступне проживання. Елемент **effect** визначає те, що має статися, якщо виконання композиції є успішним - в нашому випадку, підходяща поїздка та проживання повинні бути заброньовані. Нарешті, речення recovery[2] визначає, що має статися, якщо виконання композиції *не* вдається - поїздка або проживання не були заброньовані. Відповідно до цього елементу відновлення, якщо поїздка вже замовлена, але відбувається збій при бронюванні розміщення (наприклад, немає ніяких доступних номерів), то поїздка повинна бути скасована - в іншому випадку перша клауза речення відновлення буде порушена.

Ми хотіли б зазначити, що, в той час як мета композиції визначається за допомогою WSML-подібного синтаксису, структура і зміст мети йде істотно від цілей, що розглядаються у WSMO, які визначають набагато менш обмежені вимоги на сервіси, які мають бути виявлені (наприклад, вони не вимагають специфікацію передумов і припущень, і не містять речень відновлення). Форма цілей, прийнятих тут, необхідна, щоб представити всі вимоги, необхідні для end-to-end виявлення, композиції Веб-сервісів на функціональному рівні та рівні процесу.

3.4 Моделювання композитного сервісу

Починаючи з мети і опису існуючих сервісів, визначених раніше, передбачається, що інтегрований алгоритм виявлення і композиції вибирає набір постачальників туристичних сервісів (наприклад, сервіс Flight Booking та сервіс Hotel Booking) та об'єднує їх у виконуваний код, так що VTA може виконати його, для того, щоб взаємодіяти з цими сервісами і знайти пропозицію для запиту замовника.

Тепер ми вкажемо складену вручну композицію в термінах виконуваної специфікації інтерфейсу оркестровки для композитного сервісу. Генерування такої виконуваної оркестровки з даних сервісів Flight та Hotel є загальною метою композиції рівня процесу. Хоча ми могли б також тут використовувати семантичну специфікацію в термінах WSMO оркестровки, ми обмежимося описом композитного сервісу безпосередньо в термінах виконуваного BPEL4WS, так як синтаксис для WSMO оркестровки знаходиться в стадії розробки.

Ця BPEL4WS специфікація заявляє, насамперед, два зв'язки з двома партнерами взаємодії, а саме: сервіс з бронювання авіаквитків та сервіс бронювання готелів. Вона потім визначає деякі змінні. Видно, що може бути визначено три види змінних: вхідні змінні (які відповідні параметрам користувача, зазначених метою композиції), вихідні змінні (що визначають виходи композитного сервісу), і додаткові змінні повідомлення (використовуються для взаємодії з компонентними сервісами). Зазначимо, що вхідні змінні анотовані з посиланнями на змінні передумов, що входять в мету. Ми вимагаємо, щоб кожна BPEL4WS вхідна змінна анотувалась семантичним посиланням, яке визначає походження змінної. Вихідні змінні можуть бути анотовані аналогічно, якщо мета містить змінні постумови. У цьому випадку ми вимагаємо, щоб BPEL4WS вихідна змінна існувала для кожної змінної, визначеної в постумові мети, так що значення змінної мети визначається в кінці виконання. У нашому випадку, ми не маємо постумов мети, так що вихідні змінні не анотовані.

```
<process name="VTA">
        <partnerLinks>
            <partnerLink name="Flight" partnerLinkType="FlightRequest PLT"
                    myRole="FlightRequest _Client" partnerRole="FlightRequest_Server"/>
            <partnerLink name="Hotel" partnerLinkType="HotelRequest _PLT"
                    myRole="HotelRequest_Client" partnerRole="HotelRequest _Server"/>
        </partnerLinks>
        <variables>
                <!-- INPUT variables -->
                <variable name="start" type="xsd:string" wssem:goalReference="goal#?start"/>
                <variable name="destination" type="xsd:string" wssem:goalReference="goal#?dest"/>
                <variable name="date" type="xsd:string" wssem:goalReference="goal#?date"/>
                <variable name="client" type="xsd:string" wssem:goalReference="goal#?client"/>
                <!-- OUTPUT variables -->
```

[2] Відновлення для цілей позначають інше розширення синтаксису, визначеного в [FSPR05]. Ця функція доведе, що буде дуже корисною в контексті композиції рівня процесу, як описано в наступному розділі.

```
                <variable name="flightNo" type="xsd:integer"/>
                <variable name="hotel" type="xsd:string"/>
                <!-- MESSAGE variables -->
                <variable name="flightReq" messageType="flightRequestMsg"/>
                <variable name="flightOff" messageType="flightOfferMsg"/>
                <variable name="hotelReq" messageType="hotelRequestMsg"/>
                <variable name="hotelOff" messageType="hotelOfferMsg"/>
        </variables>
        <sequence>
             <assign>
                <copy><from variable="start"/><to variable="flightReq" part="start"/></copy>
                <copy><from variable="destination"/><to variable="flightReq" part="destination"/></copy>
                <copy><from variable="date"/><to variable="flightReq" part="date"/></copy>
                <copy><from variable="client"/><to variable="flightReq" part="client"/></copy>
            </assign>
            <invoke operation="flightRequest" variable="flightReq" partnerLink="Flight"/>
            <pick>
                <onMessage operation="flightNotAvailable" partnerLink="Flight">
                    <!-- Recovery termination -->
                </onMessage>
                <onMessage operation="flightOffer" variable="flightOff"partnerLink="Flight">
                        <assign>
                                <copy><from variable="start"/><to variable="hotelReq" part="start"/></copy>
                                <copy><from variable="destination"/><to variable="hotelReq" part="destination"/></copy>
                                <copy><from variable="date"/><to variable="hotelReq" part="date"/></copy>
                                <copy><from variable="client"/><to variable="hotelReq" part="client"/></copy>
                        </assign>
                        <invoke operation="hotelRequest" variable="hotelReq" partnerLink="Hotel"/>
                        <pick>
                              <onMessage operation="hotelNotAvailable" partnerLink="Hotel">
                                  <invoke operation="flightCancel" partnerLink="Flight"/>
                                  <!-- Recovery termination -->
                </onMessage>
                <onMessage operation="hotelOffer" variable="hotelOff"partnerLink="Hotel">
                        <invoke operation="flightConfirm" partnerLink="Flight"/>
                        <invoke operation="hotelConfirm" partnerLink="Hotel"/>
                        <assign>
                            <copy><from variable="flightOff" part="flightNo"/><to variable="flightNo"/></copy>
                             <copy><from variable="hotelOff" part="hotel"/><to variable="hotel"/></copy>
                        </assign>
                        <!-- Successful termination -->
                </onMessage>
                        </pick>
                </onMessage>
            </pick>
        </sequence>
</process>
```

Лістинг 12 - BPEL процес композитного сервісу VTA

4 Теоретичні моделі

У цьому розділі ми надаємо теоретичну основу для інтегрованого виявлення і композиції веб-сервісів. Точніше, ми надамо формальну характеристику різних елементів, відповідних входам і виходам проблеми виявлення і композиції.

4.1 Моделі для онтології предметної області

Перший елемент, який визначає вхід для задачі виявлення і композиції являє собою стандартну онтологію предметної області (див. розділ 3.1). Ця онтологія визначає основні поняття (concepts), що безпосередньо характеризують домен, і використовується в якості основи і для визначення вимог композиції, і для компонентних веб-сервісів.

Наша мета полягає в тому, щоб утримувати онтологію якомога простіше. З цієї причини, ми використовуємо дуже елементарну онтологічну мову, яка заснована на дескриптивній логіці ALN і на узагальненому ациклічному TBox [BN03]. У цій онтологічній мові (витяг з) онтологія домену лістингу 1 моделюється наступним чином: Видно, що наша формальна модель для онтології домену складається з визначення множини концептів, суперконцептів, від яких вони є похідними, і їх атрибутів. Деякі з особливостей у WSML онтології домену, наприклад, той факт, що існує асоціація один-до-одного між поїздками і ідентифікаторами поїздки, не може бути представлений в нашій моделі онтології, і, отже, виключений з рис. 4.1.

$$
\begin{aligned}
_date &\sqsubseteq \top \\
_string &\sqsubseteq \top \\
trv{\neq}Client &\sqsubseteq \forall name._string \sqcap \forall gender._string \\
trv{\neq}Location &\sqsubseteq \forall name._string \\
trv{\neq}Trip &\sqsubseteq \forall id._string \sqcap \forall date._date \sqcap \\
&\qquad \forall start.trv{\neq}Location \sqcap \forall destination.trv{\neq}Location \\
trv{\neq}TripAvailable &\sqsubseteq trv{\neq}Trip \\
trv{\neq}TripBooked &\sqsubseteq trv{\neq}Trip \sqcap \forall pax.trv{\neq}Client
\end{aligned}
$$

Рисунок 4.1 - ALN модель для онтології домену

4.2 Моделі для веб-сервісів

Веб-сервіси повинні бути змодельовані на двох рівнях абстракції, а саме, на функціональному (або можливості) рівні, і на рівні процесу (або інтерфейсу).

Модель функціонального рівня

На функціональному рівні веб-сервіс W описується:

• онтологією, яка розширює безпосередньо онтологію домену з конкретними концептами веб-сервісу (див., наприклад, онтологію Flight лістингу 2);

• набором входів, виходів, передумов і ефектів, як описано в розділі 2.1.

Формально:

Визначення 1. Опис на функціональному рівні веб-сервісу W для домену D є кортеж (Ω_W, I_W, O_W, P_W, E_W), де:

• Ω_W є розширенням онтології домену Ω_D;

• I_W являє собою набір тверджень виду $i : C$, який заявляє, що i (вхідний) індивід, який належить концепту C;

• O_W являє собою набір тверджень виду $o : C$, який заявляє, що o (вихідний) індивід, який належить до концепту C;

• P_W є умовою на Ω_W і I_W, яка визначає передумови виклику веб-сервісу;

• E_W є умовою на Ω_W, I_W і O_W, яка визначає ефекти виклику веб-сервісу.

Наприклад, у випадку сервісу польоту, опис функціонального рівня визначається таким чином:

• онтологія:

$fl\neq Flight \sqsubseteq trv\neq Trip \sqcap \forall flightNumber._string$

$fl\neq FlightAvailable \sqsubseteq trv\neq TripAvailable \sqcap fl\neq Flight \sqcap \forall seatNumber._string$

$fl\neq FlightBooked \sqsubseteq trv\neq TripBooked \sqcap fl\neq Flight \sqcap \forall seatNumber._string$

…

• вхідні дані: ?date : _date, ?start: trv#Location, ?dest: trv#Location, ?client : trv#Clients.

• вихідні дані: ?flight : fl#Flight.

• передумова відповідає припущенню, що наведене в лістингу 3.

• ефект відповідає ефекту, який з’являється в лістингу 3.

```
namespace {trv "http://www.example.org/Travel",
            fl " http :// www.example.org/BookFlight"}
webService fl#BookFlight
  capability fl#BookFlightCap
        sharedVariables {?req, ?date, ?start, ?dest, ?client}
        precondition definedBy
          ?req[
               trv#date hasValue ?date,
               trv#start hasValue ?start,
               trv#destination hasValue ?dest,
               fl#client hasValue ?client]
          memberOf fl#FlightRequest.
        assumption definedBy
             exists {?flight} ( ?flight [ trv#start hasValue ?start,
                    trv#destination hasValue ?dest,
                    trv#date hasValue ?date] memberOf fl#FlightAvailable).
        postcondition definedBy
             fl#offer (?req)[trv#date hasValue ?date,
                    trv#start hasValue ?start,
                    trv#destination hasValue ?dest,
                    trv#flightNumber hasValue fl#fn(?req),
                    fl#client hasValue ?client] memberOf fl#FlightOffer.
        effect definedBy
              fl#booking(?req)[ trv#date hasValue ?date,
                          trv#start hasValue ?start,
                          trv#destination hasValue ?dest,
                           fl#flightNumber hasValue fl#fn(?req),
                           trv#pax hasValue ?client] memberOf fl#FlightBooked.
```

Лістинг 3 – Сервіс Flight Booking: рівень можливості

Модель рівня процесу

На рівні процесу ми моделюємо веб-сервіси як *анотовані системи переходу із стану в стан*, тобто як *системи переходу із стану в стан*, які розмічені семантичними анотаціями. Системи переходу із стану в стан моделюють поведінку сервісу, в той час як онтологічні семантичні анотації описують сенс (значення) даних, з якими пов’язаний сервіс.

Система переходів із стану в стан (STS-система) моделює динамічну систему, яка може перебувати в одному стані з набору можливих *станів* (деякі з них позначені як *початкові стани*) і може розвиватися до нових станів в результаті виконання деяких *дій*. Ми розділяємо дії на *вхідні дії*, *вихідні дії* і τ. *Вхідні дії* представляють прийом повідомлень, *вихідні дії* представляють повідомлення, відправлені до зовнішніх сервісів, і τ являє собою спеціальну дію, яка називається *внутрішня дія*, яка представляє внутрішні еволюції, які не видно зовнішнім сервісам. Іншими словами, τ представляє той факт, що стан системи може розвиватися, не чинячи ніякого виведення і не обробляючи ніякі входи. *Відношення переходу* описує, як стан може розвиватися на основі входів, виходів або внутрішньої дії τ.

Визначення 2. Система переходів із стану в стан (State transition system).
Система переходів із стану в стан Σ – це кортеж $\langle S, S^0, I, O, R \rangle$ де:
S – кінцевий набір станів;

$S^0 \subseteq S$ – набір початкових станів;
I – кінцевий набір вхідних дій;
O – кінцевий набір вихідних дій;
$R \subseteq S \times (I \cup O \cup \{\tau\}) \times S$ – відношення переходу.

У анотованій системі переходів із стану в стан ми пов'язуємо з кожним станом набір *тверджень концептів* (concept assertions) і *тверджень ролей* (role assertions). Це конфігурує стан як стверджувальний (assertional) компонент системи подання знань. Твердження наведені в дескриптивній логіці і онтологія грає роль термінологічного компонента. Таким чином, *твердження концептів* являють собою формули виду $a : C$ (або $C(a)$) і заявляють, що даний індивід a належить до інтерпретації концепту C. *Твердження ролі* являють собою формули виду $a.R = b$ (або $R(a, b)$) і заявляють, що даний індивід b - це значення ролі R для a.

Визначення 3. Анотована система переходів із стану в стан (Annotated state transition system).

Анотована система переходів із стану в стан – це кортеж $\langle \Sigma, \Omega, \Lambda \rangle$, де:
Σ - це система переходів із стану в стан;
Ω – онтологія;
$\Lambda : S \rightarrow 2^{A\Omega}$ - функція анотації, де A_{Ω} - набір всіх тверджень концептів і тверджень ролей, визначених на Ω.

При відображенні веб-сервісу до анотованої системи переходів із стану в стан компонент Σ відслідковує еволюцію «статусу» (status) інтерфейсу сервісу. Точніше, стани STS-системи відповідають діям специфікації BPEL4WS або значенню змінної **ctl_state** у випадку WSMO. Онтологія Ω є тим, що визначено в описі на функціональному рівні веб-сервісу. Нарешті, функція анотації асоціює з кожним станом факти, які утримуються, коли виконання сервісу досягає цього стану. Ці факти отримуються за допомогою семантичних анотацій, які збагачують WSDL і BPEL4WS специфікації, або за допомогою фактів, які визначають стан WSMO специфікації відповідно до правил, які обговорювалися в розділі 3.2.1.

4.3 Моделі для мети композиції

Модель мети композиції повинна брати до уваги те, що мета повинна бути використана як для композиції функціонального рівня, так і для композиції на рівні процесу. З цієї причини, мета буде визначати як набір входів/виходів/передумов/ефектів, необхідних для композиції на функціональному рівні, так і формулу в певній мові мети, яка підходить для композиції на рівні процесу. Стандартна онтологія для визначення всіх цих елементів є онтологія домену, яка обговорюється в розділі 4.1.

Дамо формальне визначення умов мети на онтології домену Ω: це буде корисно для визначення не тільки передумов та ефектів, а і мови мети на рівні процесу.

Визначення 4. Умова мети (Goal condition).

Нехай $a : C$ буде твердженням концепту і $a.R = b$ буде твердженням ролі, приймаючи до уваги Ω. Тоді умова мети визначається наступним чином:
$p = a : C \mid a.R = b \mid p\,OR\,p \mid p\,\&\,p \mid NOT\,p$

Тепер ми готові визначити частину функціонального рівня вимоги композиції.

Визначення 5. Вимога композиції (composition requirement).

Нехай Ω буде онтологією домену. Вимога композиції функціонального рівня є кортеж $F = (I, O, P, E)$, де:
- I - це набір вхідних тверджень $i : C$ на Ω;
- O представляє собою набір вихідних тверджень $o: C$ на Ω;
- P є умовою мети на Ω, I, O, яка визначає передумову композиції;
- E є умовою мети на Ω, I, O, яка визначає ефект композиції.

У випадку VTA сценарію, всі компоненти, які обговорювалися раніше, легко можуть бути добуті з мети композиції, представленої в лістингу 7. Зауважимо, що умова відновлення, яка з'являється в цьому лістингу, тут не використовується. Справді, ця умова використовується тільки на рівні процесу, як це буде описано нижче.

Вимога композиції функціонального рівня є достатньою для вибору набору веб-сервісів, як це робиться у виявленні і композиції на функціональному рівні. Однак, коли рухаємося до композиції рівня процесу, цілі повинні бути розширені, щоб виразити складні вимоги, які не обмежуються умовами досяжності (як дістатися до стану, в якому обидва і політ, і готель зарезервовані). Найчастіше, цілі повинні описувати умови відновлення і переваги (як у випадку з метою у лістингу 7), або темпоральні умови (наприклад, не резервувати готель, поки ми не зарезервували політ). Ці види цілей, які поєднують переваги і темпоральні умови, можуть бути виражені в Eagle мові [DLPT02], яка може бути використана, щоб виразити умови на всій поведінці сервісу, умовах різної сили і перевагах серед різних підцілей.

Визначення 6. Eagle мета композиції $g \in G$ над умовами мети $p \in Prop$ визначається наступним чином:

g := p | g ***And*** g | g ***Then*** g | g ***Fail*** g | ***Repeat*** g |
DoReach p | ***TryReach*** p | ***DoMaint*** p | ***TryMaint*** p

Мета **DoReach** p вказує, що умова p повинна бути зрештою досягнута сильним чином, для всіх можливих недетермінованих еволюцій STS-системи. Точно так само, мета **DoMaint** g вказує, що властивість g повинна підтримуватися, незважаючи на недетермінованість. Цілі **TryReach** p і **TryMaint** g є слабшими версіями цих цілей, де потрібно план. щоб зробити «все, що можна», щоб добитися умови p або підтримати умову g , але відмова приймається, якщо неминуча. Конструкція g_1 **Fail** g_2 використовується для моделювання переваг серед цілей і відновлення після збою. Точніше, мета g_1 розглядається першою. Тільки, якщо досягнення або підтримання цієї мети не вдається, то мета g_2 використовується як відновлення або мета другого вибору. Розглянемо, наприклад, мету **TryReach** *c* **Fail DoReach** *d*. Підціль **TryReach** *c* вимагає знайти план, який спробує досягти умови *c*. У ході виконання плану, може бути досягнуто стан, з якого не вдається досягти *c*. Коли такий стан досягається, мета **TryReach** *c* не вдається і відновлення мети **DoReach** *d* розглядається. Мета g_1 **Then** g_2 вимагає спочатку досягнення мети g_1, а потім рухатись до мети g_2. Мета **Repeat** g вказує, що підціль g має бути досягнута циклічно, поки не вдається. Нарешті, мета g_1 **And** g_2 вимагає досягнення обох підцілей g_1 і g_2.

У випадку VTA сценарію, мета композиції рівня процесу складається з головної умови мети *E* та умови відновлення *R* (див. лістинг 7). У цьому випадку мета композиції рівня процесу є Eagle формула **TryReach** *E* **Fail DoReach** *R*, яка виражає вимогу, що композитний сервіс повинен спробувати (робити все можливе), щоб досягти головної умови мети *E*, і, якщо це виходить неможливо, гарантувати домогтися хоча б умови відновлення *R*.

4.4 Моделі для композитного сервісу

Останній елемент, для якого ми повинні забезпечити модель, є результат інтегрованого алгоритму виявлення і композиції, а саме композитний сервіс. У підході ми пропонуємо, ми просто моделюємо композитні сервіси як (неанотовану) систему переходів із стану в стан (визначення 2).

5 Архітектура для інтеграції виявлення і композиції

У цьому розділі ми обговоримо архітектуру для інтеграції виявлення і композиції Семантичних Веб-сервісів. Почнемо з огляду архітектури на високому рівні, потім ми зосередимося на опису інтерфейсів між різними компонентами і на визначенні основних функціональностей, що виконуються цими компонентами.

5.1 Огляд архітектури

Архітектура зображена на рис. 5.1. Вона складається з трьох модулів, які відповідають виявленню сервісу (Service Discovery - SD), композиції функціонального рівня (Functional level Composition - FLC) і композиції рівня процесу (Process Level Composition - PLC). Відповідно до цієї архітектури, підхід працює у двох фазах. Під час першої фази SD і FLC чергуються, щоб знайти набір веб-сервісів, що, як тільки відбувається їх композиція, вони здатні задовольнити ціль клієнта. Друга фаза вводиться один раз, коли цей набір веб-сервісів був знайдений: PLC викликається, щоб генерувати реальний виконуваний код, який реалізує композицію. Тепер ми дамо більш докладний опис трьох блоків в архітектурі.

Компонент «Service Discovery».

Мета: знайти Веб-сервіс, який відповідає запиту.

Завдання: посередництвом доступів до каталогу за рахунок кешування існуючих результатів і зіставлення нових запитів з уже виявленими сервісами.

Вхід: запит на виявлення.

Використовує: описи можливостей Веб-сервісу в каталозі.

Вихід: Веб-сервіс, який відповідає запиту (чи відмова, якщо Веб-сервіс не знайдено).

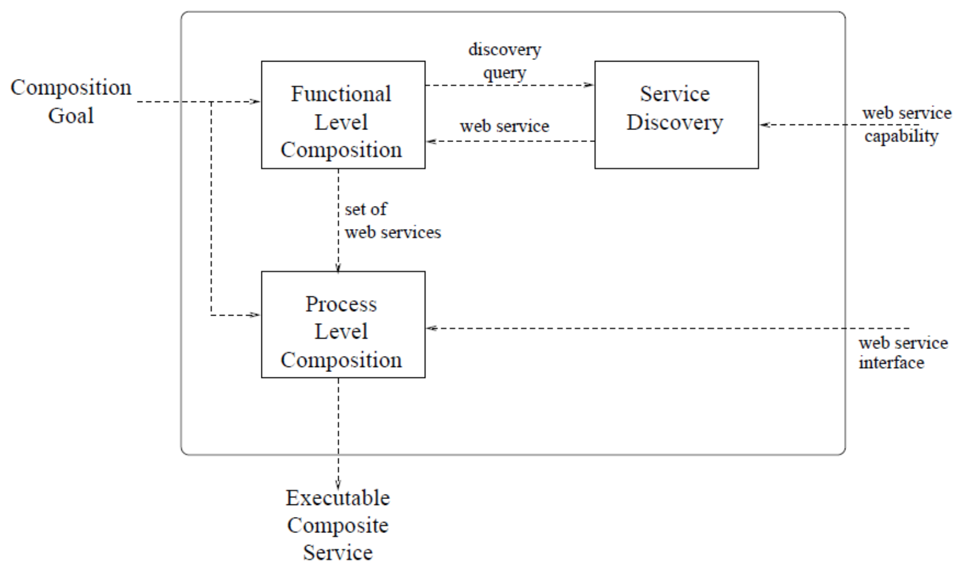

Рисунок 5.1 – Схема інтеграції виявлення і композиції

Компонент «Functional level Composition».

Мета: знайти набір Веб-сервісів, які відповідають меті композиції.

Завдання: поступово трансформувати мету композиції в набір Веб-сервісів, зіставляючи його. Методи прямої/зворотної побудови ланцюжка, які запропоновані в [CFB04], можуть тут бути прийняті.

Вхід: мета композиції.

Вихід: набір Веб-сервісів, які відповідають меті (або підцілі).

Компонент «Process level Composition».

Мета: побудувати виконуваний композитний Веб-сервіс.

Завдання: враховуючи набір Веб-сервісів, які відповідають меті композиції, генерувати виконуваний код, що, як тільки виконується, взаємодіє з компонентними сервісами і досягає мети.

Вхід: набір Веб-сервісів, мета композиції.

Використовує: описи інтерфейсів Веб-сервісів в каталозі.

Вихід: виконувані композитні веб-сервіси.

Рис. 5.1 зосереджує увагу на потоку даних між різними компонентами. Потік управління є складним у зв'язку з необхідністю управління невдачами в досягненні композиції і відкатами попередніх варіантів. Розглянемо, наприклад, компонент SD: він, як правило, ідентифікує кілька Веб-сервісів, які в змозі відповідати заданому запиту на виявлення, однак тільки один з них повертається в FLC. У разі, якщо компонент FLC не може знайти відповідний набір Веб-сервісів, то контроль може бути повернений до компоненту SD та інший Веб-сервіс, який відповідає цілі, може бути розглянутий. Аналогічним чином, якщо PLC не здатний генерувати виконуваний процес, що задовольняє мету даного певного набору Веб-сервісів, управління може повернутися до компоненту FLC, який може обчислити альтернативний набір Веб-сервісів, що вирішують ту ж проблему FLC. Цікаве питання полягає у визначенні відповідної інформації, яка буде відправлена назад від PLC до FLC і від FLC до SD для того, щоб направити процес повернення.

Питання залишається відкритим з нинішньою архітектурою, яка повинна вирішуватися в майбутній роботі. Проблема полягає у визначенні критерію для орієнтування компонентів SD і FLC у виборі набору Веб-сервісів, так що б отримати високоякісні композитні сервіси. Проблема тут у тому, що в той час, як якість композиції можна оцінити тільки після того, як виконуваний сервіс був створений компонентом PLC, ця якість відображує ті вибори (choices), які зробили в FLC і SD.

Нарешті, цікаве розширення запропонованої архітектури складається з урахування нефункціональних вимог, таких як QoS, витрати, безпека і т. д., у відборі і композиції сервісів.

5.2 Визначення інтерфейсів

У цьому розділі ми визначимо в більш точний спосіб інтерфейсів між різними блоками, описаними на рис. 5.1. Ми почнемо з визначення деяких основних типів даних, які ми надалі не описуватимемо в цьому документі.

URI

Опис: унікальні ідентифікатори (наприклад, URL) для Веб-сервісів і пов'язаних з ними BPEL4WS процесів.

OntologyExpression

Опис: формула, яка визначає умову на мові ALN.

EaGLeExpression

Опис: формула, яка визначає умову в Eagle мові.

Визначимо тепер один з ключових типів даних, а саме онтологію.

Онтологія

Опис: онтологія на мові ALN.

Методи:

• *getConcepts()*: *Set of Objects* - повертає набір концептів онтології

• *getConceptDefinition(c:Object)*: *OntologyExpression* - повертає визначення концепту.

Різні інші типи даних будуються на вершині типу даних Онтологія. Найпростіший з них є твердження (assertion).

Твердження (assertion**)**

Опис: твердження форми *i: C*

Атрибути:

• *o: Ontology* - основна онтологія

• *i: Object* - індивід *i*

• *c: Object* - концепт *C* для індивіда; *c* належить до *o.getConcepts ()*.

Ми готові визначити типи даних для виявлення і для вимог композиції.

DiscoveryGoal

Опис: вимога виявлення

Атрибути:

- *DO: Ontology* - онтологія домену
- *I*: *Set of Assertion* - вхідні твердження на *DO*
- *O*: *Set of Assertion* - вихідні твердження на *DO*
- *P*: *OntologyExpression* - передумова мети
- *E*: *OntologyExpression* - ефект мети

CompositionGoal

Опис: вимога композиції

Атрибути:

- *DO*: *Ontology* - онтологія домену
- *I: Set of Assertion* - вхідні твердження на *DO*
- *O*: *Set of Assertion* - вихідні твердження на *DO*
- *P: OntologyExpression* - передумова мети
- *E: OntologyExpression* - ефект мети
- *G: EaGLeExpression* - мета композиції рівня процесу

Тепер визначимо тип даних, що описує Веб-сервіс на функціональному рівні.

WebService

Опис: визначення Веб-сервісу на функціональному рівні

Атрибути:

- *id: URI* - унікальний ідентифікатор веб-сервісу
- *DO*: *Ontology* - онтологія домену
- *SO*: *Ontology* - онтологія сервісу
- *I*: *Set of Assertion* - вхідні твердження
- *O*: *Set of Assertion* - вихідні твердження
- *P: OntologyExpression* - передумова сервісу
- *E: OntologyExpression* - ефект сервісу

Наступні структури даних відповідають системам переходів із стану в стан і анотованим системам переходів із стану в стан, відповідно.

STS

Опис: визначення системи переходів із стану в стан, пов'язане з (компонентним або композитним) сервісом

Атрибути:

- *S*: *Set of Objects* - стани STS
- ...

Зверніть увагу, що ми не описуємо систему переходів із стану в стан в деталях, ми тільки явно описали один з її компонентів, а саме набір станів. Зауважимо, що з концептуальної точки зору він складається з компонентів, описаних у визначенні 2, однак ефективні кодування таких структур, таких, як ті, що експлуатуються в [PTBM05b, PMBT05], необхідні, щоб тримати невеликий розмір об'єктів і забезпечити ефективну композицію рівня процесу.

ASTS

Опис: визначення анотованої системи переходів із стану в стан пов'язане з компонентним сервісом

Атрибути:

- *id* : *URI* - унікальний ідентифікатор інтерфейсу веб-сервісу
- *sts : STS* - основна STS
- *o : Ontology* - онтологія сервісу
- *l* : *Map of Object to OntologyExpression* - функція маркування; домен складається із станів у *sts.S*

5.3 Основні функціональності

Тепер визначимо основні функціональності для нашої архітектури інтегрованого виявлення та композиції. Насамперед, ми визначимо деякі функціональні можливості, які необхідні для доступу до опису даного сервісу. Зокрема, наступні дві функції дають доступ до моделі рівня можливостей і рівня процесу Веб-сервісу.

getFunctionalDescription (ws: URI): WebService
Опис: отримує опис Веб-сервісу функціонального рівня (або рівня можливостей)
Параметри:
• *ws: URI* - унікальний ідентифікатор Веб-сервісу
Повертає:
• *WebService* - можливості Веб-сервісу

getProcessDescription (ws: URI): ASTS
Опис: отримує опис Веб-сервісу рівня процесу
Параметри:
• *ws: URI* - унікальний ідентифікатор Веб-сервісу
Повертає:
• *ASTS* - інтерфейс Веб-сервісу рівня процесу

Зазначимо, що попередні дві функції відповідають за те, щоб генерувати моделі, які ми використовуємо для представлення Веб-сервісів з опублікованого опису цих Веб-сервісів. Оскільки опублікований опис Веб-сервісів може бути на різних мовах (наприклад, WSMO хореографія або BPEL4WS для опису рівня процесу), ці функції мають різні реалізації, залежно від конкретної мови (ми будемо мати, наприклад, getProcessDescriptionBPEL і getProcessDescriptionWSMO).

Наступна функція описує основну функціональність компонента виявлення.

search(g: DiscoveryGoal): Set of URI
Опис: отримує набір сервісів, які відповідають заданій цілі виявлення
Параметри:
• *g: DiscoveryGoal* - мета виявлення
Повертає:
• *Set of URI* - набір Веб-сервісів, кожен з яких відповідає меті виявлення; кожен сервіс описується своїм унікальним ідентифікатором

Примітка: Набір сервісів не обов'язково буде вичерпним, тобто, там можуть бути сервіси, які відповідають цілі, що не повертаються функцією. Точні відомості про те, які відповідні сервіси повертаються, залежить від конкретної реалізації компоненту виявлення.

Наступне визначення відповідає блоку композиції функціонального рівня.

FLC(g: CompositionGoal): Set of URI
Опис: знаходить набір сервісів, які спільно задовольняють мету композиції
Параметри:
• g: CompositionGoal - мета композиції
Повертає:
• *Set of URI* - набір Веб-сервісів таких, які, у поєднанні відповідним чином, задовольняють мету композиції на функціональному рівні; кожен сервіс описується унікальним ідентифікатором

Нарешті, функція для композиції рівня процесу.

PLC(stss: Set of ASTS, g: CompositionGoal): STS

Опис: генерує композицію рівня процесу набору анотованих систем переходів із стану в стан

Параметри:

• *stss: Set of ASTS* - набір анотованих систем переходів із стану в стан

• *g: CompositionGoal* - мета композиції

Повертає:

• *STS* - система переходів із стану в стан, яка реалізує композитний сервіс

Примітка: набір, якщо URI повертається функцією FLC, повинен бути перетворений в набір STS, перш ніж він може бути переданий функції PLC.

Остання функція, яку ми наведемо тут, відповідає розгортанню композитного сервісу.

deploy(sts: STS)

Опис: розгортає систему переходів із стану в стан для композитного сервісу

Параметри:

• *sts: STS* - система переходів із стану в стан, яка реалізує композитний сервіс

Примітка: реалізація цієї функції залежить від кінцевої мови для композитного сервісу. З цієї причини, ми можемо мати різні реалізації цієї функції, такі як deployBPEL, deployJAVA і т.д.

6 Висновки

Очікується, що автоматичне створення виконуваного Веб-сервісу, починаючи з бажання користувача, має великий вплив в галузі електронної комерції та інтеграції корпоративних додатків, так як може дозволити динамічну і масштабовану співпрацю між різними системами і організаціями.

Важливим кроком на шляху динамічної і масштабованої інтеграції є розуміння того, як виявлення та композиція можуть бути використані для створення виконуваного веб-сервісу. Схема, запропонована в розділі 5, показує логічний взаємозв'язок між компонентами цього інтегрованого підходу, і як можна об'єднати їх, щоб знайти веб-сервіс, який відповідає меті, якщо необхідно, то уточнити мету і, нарешті, композувати веб-сервіси в унікальний виконуваний процес.

У цьому документі ми зосередилися на описі підходящих мов і теоретичних моделей для такої інтеграції виявлення і композиції. Більше того, ми запропонували архітектуру, яка підтримує цю інтеграцію. Майбутня робота буде включати реалізацію інтегрованого виявлення веб-сервісу і алгоритму композиції, заснованої на концептах і на архітектурі, описаних в цьому документі. Реалізацію прототипу планується завершити протягом 30 місяців.

6.1 Розширення архітектури: репутація

У цьому розділі ми обговоримо розширення архітектури інтегрованого виявлення і композиції, обговорених раніше, яка враховує репутацію аспектів у виборі веб-сервісів, які будуть підлягати композиції.

У відкритому середовищі, де зловмисники можуть рекламувати помилкові сервісні можливості, використання репутації сервісів є перспективним підходом для пом'якшення таких атак. Сервіси, які погано себе поводять, отримують погану репутацію (за повідомленням розчарованих клієнтів) і можуть бути уникнуті іншими клієнтами. Механізми репутації сприяють поліпшенню глобальної ефективності системи в цілому, тому що вони знижують стимул обдурити [Bir01]. Дослідження показують, що покупці серйозно враховують репутацію продавця, коли розміщують свої ціни в інтернет-аукціонах [HW01]. Більш того, було доведено, що в деяких випадках механізми репутації можуть бути розроблені таким чином, що в інтересах кожного учасника повідомляти правильну інформацію репутації (стимулювання сумісних сервісів репутації) [JF04]. Крім того, механізми репутації можуть бути реалізовані в захищеному режимі [JF03].

Детальний опис існуючих механізмів репутації та їх застосування в семантичних веб-сервісах виходить за рамки цього документа. Зацікавлений читач може знайти докладну інформацію в Deliverable 2.4.9 [JFB05]. Тут ми наводимо простий підхід для інтеграції механізмів репутації в процес вибору сервісу.

Ми забезпечимо веб-сервіс репутації, що дозволяє запитувати репутацію інших сервісів, а також подавати звіти. Веб-сервіси репутації будуть мати розширювану архітектуру, що дозволяє плагінити і розгортати різні конкретні механізми репутації для різних сценаріїв прикладу.

Інтеграція механізмів репутації в процес виявлення дозволяє відфільтрувати сервіси, які мають погану репутацію, і ранжувати відповідні сервіси, згідно до їх репутації. Таким чином, ми забезпечимо оболонку для компонента виявлення, яка спершу направляє запит в стандартний компонент виявлення, а потім отримує репутацію виявлених сервісів шляхом доступу до веб-сервісу репутації. Ґрунтуючись на репутації виявлених сервісів, деякі сервіси можуть бути видалені з результату (якщо репутація нижче заданого порогу) або порядок виявлених сервісів в результаті може бути змінений відповідно до репутації (сервіси з більш високою репутацією на першому місці). Рис. 6.1 ілюструє наш підхід.

Цей підхід має ту перевагу, що не вимагає ніяких змін в архітектурі інтегрованого виявлення та композиції на рис. 5.1. Механізми репутації добре інкапсулюються у веб-сервіс репутації. Компоненти виявлення і композиції повністю функціональні без веб-сервісу репутації, який може бути інтегрований за допомогою просто інсталяції вищезгаданої оболонки для компонента виявлення.

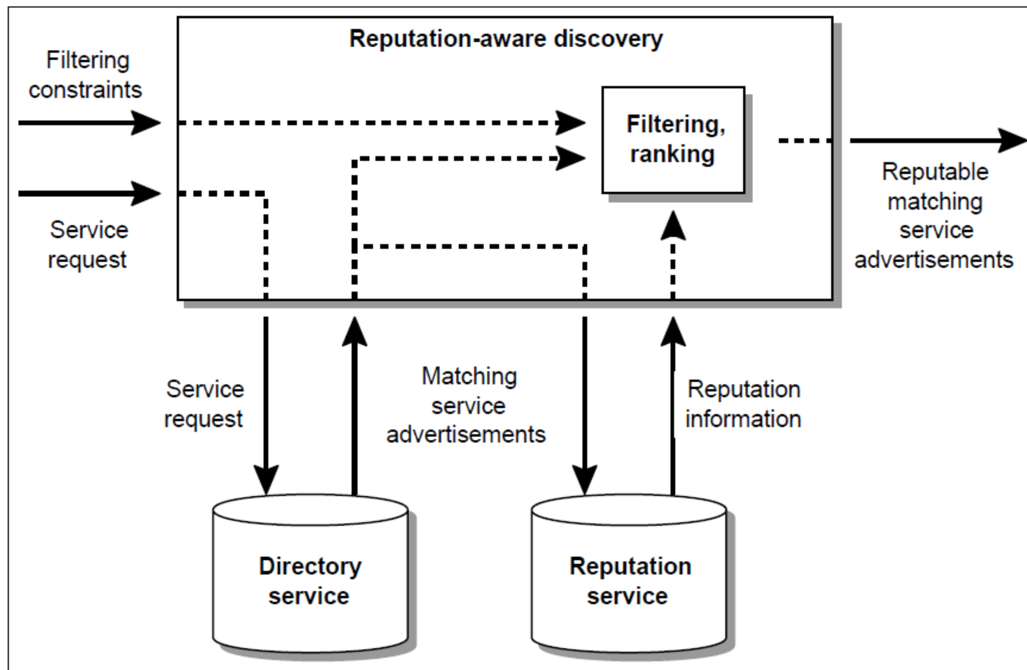

Рисунок 6.1 - Оболонка для фільтрів виявлення сервісів і рангів відповідних оголошень сервісів відповідно до їх репутації

Література

[ACD+03] T. Andrews, F. Curbera, H. Dolakia, J. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weeravarana. Business Process Execution Language for Web Services (version 1.1), 2003.

[Bir01] A. Birk. Learning to Trust. In R. Falcone, M. Singh, and Y.-H. Tan, editors, *Trust in Cyber-societies*, volume LNAI 2246, pages 133–144. Springer-Verlag, Berlin Heidelberg, 2001.

[BN03] F. Baader and W. Nutt. Basic Description Logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors, *The Description Logic Handbook*, pages 43–95. Cambridge University Press, 2003.

[BS03] Egon B¨orger and Robert St¨ark. *Abstract State Machines*. Springer, 2003.

[CFB04] I. Constantinescu, B. Faltings, and W. Binder. Typed Based Service Composition. In *Proc. WWW2004*, 2004.

[Coa03] The OWL Services Coalition. OWL-S: Semantic Markup forWeb Services. In *Technical White paper (OWL-S version 1.0)*, 2003.

[DLPT02] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a Language for Extended Goals. In *Proc. AAAI'02*, 2002.

[FSPR05] Dieter Fensel, James Scicluna, Axel Polleres, and Dumitru Roman. Ontology-based choreography and orchestration of WSMO services. Deliverable d14v0.2, WSMO, 2005. Available from http://www.wsmo.org/TR/d14/v0.2/20051008/.

[HW01] D.E. Houser and J. Wooders. Reputation in Internet Auctions: Theory and Evidence from eBay. University of Arizona Working Paper #00-01, 2001.

[JF03] R. Jurca and B. Faltings. An Incentive-Compatible Reputation Mechanism. In *Proceedings of the IEEE Conference on E-Commerce*, Newport Beach, CA, USA, 2003.

[JF04] R. Jurca and B. Faltings. "CONFESS". An Incentive Compatible Reputation Mechanism for the Online Hotel Booking Industry. In *Proceedings of the IEEE Conference on E-Commerce*, San Diego, CA, USA, 2004.

[JFB05] Radu Jurca, Boi Faltings, and Walter Binder. Reputation mechanism. Knowledge Web Deliverable D2.4.9, 2005.

[Lar04a] Ruben Lara. Definition of semantics for web service discovery and composition. Knowledge Web Deliverable D2.4.2, 2004.

[Lar04b] Ruben Lara. Semantic requirements for web services description. Knowledge Web Deliverable D2.4.1, 2004.

[LKMR+05] Holger Lausen, Uwe Keller, Francisco Martin-Recuerda, Jos de Bruijn, , and Michael Stollberg. A conceptual and formal framework for semantic web services. Knowledge Web Deliverable D2.4.5, 2005.

[MSZ01] S. McIlraith, S. Son, and H. Zeng. SemanticWeb Services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[PMBT05] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. IJCAI'05*, 2005.

[PSK02] M. Paolucci, K. Sycara, and T. Kawamura. Delivering Semantic Web Services. In *Proc. WWW2003*, 2002.

[PTBM05a] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. An approach for the automated composition of BPEL processes. In *Proc. Workshop on WWW Service Composition with Semantic Web Service (WSCOMPS 2005)*, 2005.

[PTBM05b] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *Proc. ICWS'05*, 2005.

[SVMR05] A. Sheth, K. Verna, J. Miller, and P. Rajasekaran. Enhacing Web Service Descriptions using WSDL-S. In *EclipseCon*, 2005.

[WSM05] Web service modeling ontology (WSMO), June 2005. W3C member submission. Available at http://www.w3.org/Submission/WSMO/.