Toward a framework for automated service composition and execution : E-tourism Applications

Pathathai Na Lumpoon

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 2013

Présentée par

Pathathai Na-Lumpoon

Thèse dirigée par **Ahmed Lbath**
et codirigée par **Marie-Christine Fauvet**

préparée au sein **Laboratoire d'Informatique de Grenoble**
et de **Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII)**

Vers une plateforme dediee a la composition automatique et l'execution de services: Applications E-Tourisme

Thèse soutenue publiquement le **12 May 2015**,
devant le jury composé de :

Prof. Marlon DUMAS
University of Tartu, Estonie, Président
Prof. Abdelaziz BOURAS
University of Lyon 2, Rapporteur
Prof. Nadine CULLOT
University of Bourgogne, Dijon, Rapporteur
Prof. Marie-Christine Fauvet
Joseph Fourier University of Grenoble, Co-Directeur de thèse
Prof. Ahmed LBATH
Joseph Fourier University of Grenoble, Directeur de thèse

"We are what we repeatedly do. Excellence then, is not an act, but a habit."

Aristotle

Abstract

Recently, distributed computing systems based on context awareness have been proposing in several domains such as healthcare, logistics and tourism. The study described in this thesis is a part of a broader project of E-Tourism system that provides nomad user, context-aware personalized services. The work of this thesis focuses on the issues raised by web service composition and execution. Web service composition brings benefits of reusing existing services to synthesize the new resulting service. The way to create web service composition normally happens at design time, but this limits choices of services. This thesis presents a novel approach of automated web service composition and execution framework. Our approach aims to compose service operations that fulfill mobile users' requirements expressed in propositional logics and execute the resulting service based on aggregating multi service components. In the framework, we have introduced our planning algorithm based on abstract goal states to search and connect to service operations, by mean of service operation annotations, for an abstract plan. The abstract plan is expected for workflow model of sequencing, paralleling and conditioning among service operations. However, the generated workflow is not in an executable form, this is why we perform the transformation of the workflow into an executable business process. To achieve the business process execution, we defined context based BPMN model for mapping the abstract plan to BPMN semantics. We also propose a new planning algorithm and exploited planning-as-model-checking approach to validate the semantic BPMN model whether it is Well-formed and Well-defined BPMN process. Finally, we implemented the automated service composition and execution framework system in Java platform as a proof of concept. We developed the logical composition and, transformation and validation BPMN algorithms written in Prolog. We have integrated these algorithms into the whole system. As future work, we plan to extend our current work and take into account more complex user's scenarios that explicitly and implicitly express other pattern controls and we will also consider the scenarios required for stateful web services.

Résumé

Les systèmes de services contextualisés ont connu un grand essor ces dernières années dans des domaines variés tels que la santé, la logistique ou bien le tourisme. Cette thèse s'intègre dans un projet plus global, nommé E-Tourism, qui vise à réaliser une plateforme de fourniture de services sensibles au contexte à des utilisateurs en situation de mobilité. Le travail de cette thèse se focalise sur la problématique de composition de services web et de leur exécution. La composition de services web permet la réutilisation de services existants afin d'en faire une synthèse répondant au besoin de l'utilisateur. Cette composition se fait en général au niveau de la phase conceptuelle ce qui limite le choix de services potentiels. Cette thèse présente une nouvelle approche de la composition de services Web automatisé et une plateforme d'exécution. Notre approche vise à composer les opérations de services qui répondent aux besoins des utilisateurs mobiles exprimées dans une logique propositionnelle et exécuter le service composé en agrégeant plusieurs services composants. Nous avons proposé une plateforme d'exécution de services et nous avons introduit un nouvel algorithme de planification intégré à cette plateforme. Nous avons défini un contexte basé sur le modèle BPMN. Afin de valider notre modèle sémantique nous avons utilisé une approche de type model-checking. Enfin, nous avons réalisé un prototype sous forme d'une plateforme de composition de services et d'exécution automatique en Java. Nous avons implémenté les règles d'inférence et les modèles BPMN en prolog. Enfin, nous avons intégré ces algorithmes dans l'ensemble du système. En perspectives, nous prévoyons d'étendre notre travail actuel et prendre en compte des scénarios plus complexes nécessitant des services web dynamiques.

Acknowledgements

This dissertation would not have been possible without the support of many great people, to whom I express my gratitude.

First of all, I would like to thank Professor Ahmed Lbath, my supervisor, for giving me the opportunity to pursue a PhD and be part of MRIM team, LIG, University of Grenoble. His support and guidance enabled me to complete this project. I am specially thankful to Professor Marie Christine-Fauvet, my supervisor, for her valuable all-round support of my doctoral research work. I would have not made it this far without her invaluable and generous investment into my research.

I am also grateful to European Commission under Sustainable e-Tourism, Erasmus Mundus Action 2 and Chiang Mai university who have been providing me with financial support during my period of my doctoral research. I also appreciate Professors and colleges from MRIM team who have been advising and supporting me during my 4-year research.

I would like to greatly thank Professor Nadine Cullot and Professor Abdelaziz Bouras for reviewing the manuscript and providing me with valuable comments. I wish to thank Krisdee Promubol for correcting English grammars of the manuscript.

I would like to especially thank my project mates. As my research is part of E-tourism project, I have been working with four PhD students, Isaac Caicedo Castro, Mu lei, Teerawat Kamnerdsiri and Uyanga Sukhbaatar, and one master student, Sanjay Kamath. We were supporting each other and helping each other to the best of our research works.

My PhD life would not be meaning and colorful without all my friends of the LIG, with which I shared these year in particular. Thanks to you all Sarah, Quang, David, Mateuze, Fred and Nadia.

Last but not least, I want to thank you my family, my father, my mother and my sister, for having always been near me in these last years of study. Without their supports this study would have never come into existence. I also want to thank you my boyfriend, Pree Thiengburathum, for his supporting and effort by keep inspiriting and motivating me on both work and personal life...

Contents

List of Figures

List of Tables

Abbreviations

ADL	**A**ction **D**escription **L**anguage
AI	**A**rtificial **I**ntelligent
ALAN	**A**xiomatization **LA**nguage for **A**gents
B2B	**B**usiness-to-**B**usiness
B2C	**B**usiness-to-**C**ustomer
BPEL	**B**usiness **P**rocess **E**xecution **L**anguage
BPMN	**B**usiness **P**rocess **M**odel **N**otation
COM	Microsoft **C**omponent**O**bject **M**odel
COS	**C**omposition and **O**rchestration **S**ystem
DCOM	**D**istributed **C**omponent **O**bject **M**odel
DS	**D**iscovery **S**ystem
ESB	**E**nterprise **S**ervice **B**uses
FLUX	**FLU**ent E**X**ecutor
HTN	**H**ierarchical **T**ask **N**etwork
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
MBP	**M**odel **B**ased **P**lanning
OMG	**O**bject **M**anagement **G**roup
OWL-S	**W**eb **O**ntology **L**anguage for **S**ervices
PDDL	**P**lanning **D**omain **D**efinition **L**anguage
QM	**Q**uery **M**anagement system
QoS	**Q**uality of **S**ervices
REST	**RE**presentational **S**tate **T**ransfer
RMI	**R**emote **M**ethod **I**nvocation
RPC	**R**emote **P**rocedure **C**all

SAWSDL	**S**emantic **A**nnotations for **WSDL** and XML Schema
SHOP2	**S**imple **H**ierarchical **O**rdered **P**lanner 2
SOA	**S**ervice-**O**riented **A**rchitecture
SOA	**S**ervice-**O**riented **C**omputing
SOAP	**S**imple **O**bject **A**ccess **P**rotocol
STRIPS	**ST**anford **R**esearch **I**nstitute **P**roblem **S**olver
UIQM	**U**ser **I**nteraction **Q**uery **M**anagement
W3C	**W**orld **W**ide **W**eb **C**onsortium
WfMC	**W**ork**f**low **M**anagement **C**oalition
WSDL	**W**eb **S**ervice **D**escription **L**anguage
WSSL	**W**eb **S**ervice **S**pecification **L**anguage
XPDL	**X**ML **P**rocess **D**efinition **L**anguage
XML	E**X**tensible **M**arkup **L**anguage
YAWL	**Y**et **A**nother **W**orkflowition **L**anguage

Chapter 1

Introduction

Contents

Abstract. In this opening Chapter of the thesis we discuss distributed computing of software system regarding to its trend and important to business cooperation and technologies used to build such distributed systems. In this dissertation, we will tackle a problem of the composition of existing services. The service composition problem is sourced from the Service Composition and Execution system, which is a part of E-Tourism system. The E-Tourism system and motivating scenarios from tourism domain are introduced in this Chapter. In this Chapter we also present a compress view of our work on the Service Composition and Execution Framework, which consequently motivated and laid the foundation for the major contributions we committed in this dissertation.

Over the past decades, the increasing distributed computing of software system has led to enormous rise in application complexity. For example, distributed computing applications in domains such as health-care, logistics and tourism. The

study described in this thesis is a part of a broader project of E-Tourism system. The distributed computer program is assembled using a combination of services from a variety of applications. The applications vary from difference of operating systems, middleware platforms, and programming languages. The idea of distributed computing is originated from the need of cooperation among software components. Software components vary from computer programs to service applications inter-acting across enterprises.

One example of distributed systems is Business-to-Business (B2B) system. The B2B describes commercial transaction between businesses such as between wholesalers and retailers. To design B2B interactions, the issues in communication, content and business process need to be resolved. The B2B communication assures that among remotely located partners are able to exchange messages. The content due to the use of standard over B2B guarantee no ambiguous information is exchanged among parters. The business process in B2B structures and measures set of activities to produce a specified output for a particular customer or market. Technologies, such as COM[1], Java RMI[2] and web services, are possible choices to implement B2B and other distributed systems. All these technologies rely on Service-Oriented Architecture (SOA)[3].

This dissertation copes the problems of Service composition. Service composition is one of the foundation technologies within SOA. It can be a part of and/or new application which needs to compose a sequence, condition and loop among the component services to fulfill the user's need. In this PhD, we have selected web services to reuse existing services for composing new services (or application) with higher functionality. The reason why we have selected web services in service composition problem is threefold:

- SOA-based applications are quickly tuned and adjusted to new business requirements. As it allows for rapid development of new application which results service compositions, especially when all the necessary components are already available.

[1]The Microsoft Component Object Model (COM) is a platform-independent, distributed, object-oriented system for creating binary software components that can interact.

[2]Java Remote Method Invocation (Java RMI) enables the programmer to create distributed Java technology-based to Java technology-based applications possibly on different hosts.

[3]Service-oriented architecture is a design pattern based on distinct pieces of software providing application functionality as services to other applications via a protocol.

- The proliferation of web service development. Web services have become the dominant choice of implementation for SOA-based system. Statistics from programmableWeb[4] said that the number of published web services have been increasing continuously from 2005 to 2013. The recent number has been recorded for over 12,213 web services. This indicates the trend of applications realizing web services keeps increasing.

- This dissertation deals with the service composition and execution part of the E-Tourism system (see Figure 1.1). The E-Tourism system aims at providing context-aware personalized services for nomad users. The objective of the system is to locate and deliver the right service to the right person, at the right time and location, with the appropriate rendering [Lba05, LLK+11, LLCC+13]. We will describe in more details the E-Tourism system in the next section.

1.1 E-Tourism system Architecture

In this research we aim to cope the problem of (1) composing service operations that fulfil users' requirements expressed in propositional logics (or called goal state(s)) (i.e., BookingTableReservation state and Direction state for the user query *I want to book a table for 2 people at the finest restaurant at 8pm. in the city, and the direction to the restaurant.*") and (2) executing the resulting service solution based on aggregating multi service components (i.e., FromCoordinatesToCity operation, FindFinestRestaurant operation, BookRestaurant operation and GetDirection operation) back to the user. Therefore, we proposed a new approach for automated service composition and execution to tackle the aforementioned illustrated problem. The approach is part of the system sketched in Figure 1.1. This system is designed to provide mobile users with services [LLK+11, LLCC+13]. For instance, services to book a room, or reserve a table in a restaurant located in a certain city, etc. The role of each module, and the flows of information are detailed as follows:

1. User interaction and query management (UIQM) module aims at managing user connections and getting queries submitted by users and sent using their

[4] http://www.programmableweb.com/

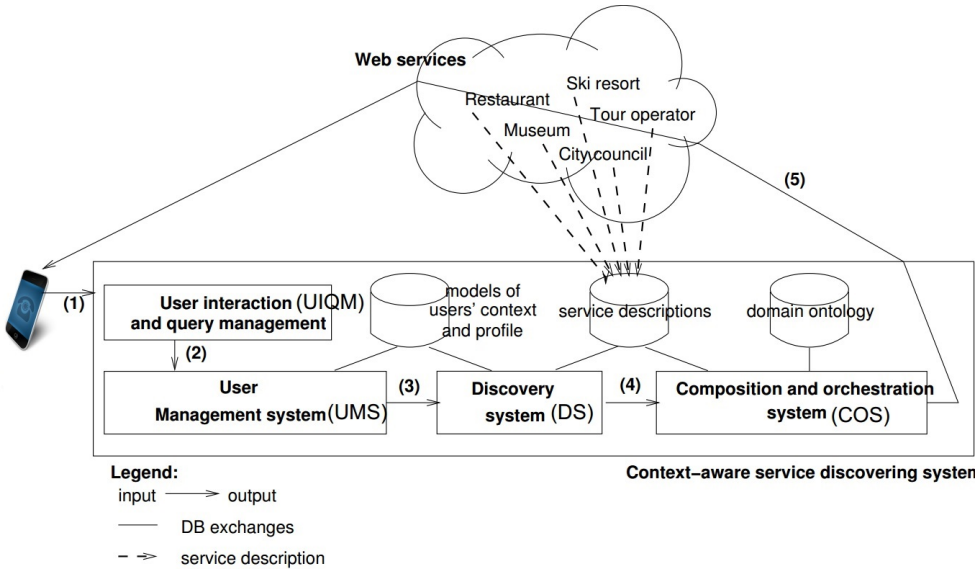


FIGURE 1.1: Architecture of a system for discovering, execution and composition of services for mobile users [LLK+11]

mobile device. Users' queries, identifications, and context information are received by this module in the data flow (1). This module extracts from the query all necessary information for the discovery, composition, and execution of services.

2. The module User management System (UMS) is in charge of managing users' context and profile with respect of their privacy. This module receives the users' identifications in the data flow (2). Thereafter, this module sends forward by means of the data flow (3) users' queries, context and class of profile.

3. Discovery system (DS) is responsible for finding service operations that fulfil users' requirements expressed in free text queries. Given a user's query (received in the data flow (3)), what are the services which may potentially meet the user's needs expressed in such query? The services which fulfil the user's needs are sent, in the data flow (4), to the next module.

4. Eventually the module Composition and orchestration system (COS) is a module which is in charge of automated service composition and execution. The list of service operations and users' query in the data flow (4) pass to this module. Its aim is to compose operations, offered by discovered services, to fulfill users' query. The result is a composite service, which is executed later by an orchestration engine. During the execution, if the users chose not to disclose their privacy then some parameters from their profile and context

may be missing. In this case, the orchestration engine will interact to the users to ask for the missing parameters. Finally, the resulting composite service in the data flow (5) is sent back to the users.

This research focuses on the issues raised by the *Composition and orchestration system* in the fourth above mentioned item (The other modules are handles by other PhD students).

1.2 Motivating Examples

Alice is an American tourist visiting Paris in France. She has forgotten to search hotel rooms for tonight. Thus, she picks up her smartphone and accesses the system above mentioned (see Figure 1.1), and issues the query: I want to search for apartment rooms available from tonight to 05/06/2014. The Query Management system (QM) analyzes the query to retrieve initial parameters: CheckIn = "tonight"[5] , CheckOut = "05/06/2014" and RoomType = "apartment" and goal parameter: RoomAvailables. Moreover, the system captures Alice's context information, namely: coordinates = "48.2167° N, 2.3332° E" and date = "1/06/2014". Besides, the system has the following information about Alice's profile: name = "Alice", citizenship="USA", travelPurpose="tourism", gender="female". For privacy reason, Alice wants to hide her location so the User Interaction and Query Management (UIQM) module dose not disclose her location information out from her context.

Thereafter, the Discovery system (DS) searches services for searching available rooms. It gives to the Composition and Orchestration System (COS) a ranked list of candidate services for searching available rooms. The service which has the highest rank in this list contains the following operation: SearchAvailableRooms, this operation receives as parameters the destination location, the number of nights, the number of guests, the type of room, the date and time to check-in and check-out. As a result, the operation returns a list of available rooms.

In this case, the COS interacts with Alice to ask the missing values for the parameters Location and NumberOfGuest. COS executes this operation by using Alice's

[5]where tonight is the 01/06/2014 according to Alice's smart phone.

context information and Alice successfully searches for apartment rooms available in Paris.

At 4 PM, she wants to book a table at the finest restaurant in the city, and the direction to get there. Once again, she uses the same system and submits the query: I want to book a table for 2 people at the finest restaurant in the city, and the direction to the restaurant. At this time, Alice's profile has not been changed, and the system captures the following Alice's context information: coordinates = "48.8567° N, 2.3508° E" and date = "1/06/2014".

This query has two requirements, then the User Interaction and Query Management (UIQM) module splits this query in two subqueries. Therefore, the first sub-query submitted to the DS is I want to book a table at the finest restaurant in the city. The second sub-query issued to the DS is the direction to the restaurant. The DS shall send to the COS two ranked lists of candidate services, which correspond with each subquery. From the list of candidate services that may fulfil the first subquery, the one which has the highest rank contains the following operations:

- FindFinestRestaurant: This operation receives as a parameter the name of the city where the user is looking for the finest restaurant. The operation returns the name and the address of the restaurant.
- BookRestaurant: This operation receives as parameters the restaurant name, the number of persons, and the user's name and telephone number. As a result, the operation returns a confirmation whether the table has been booked or not.

In another ranked list of candidate services that fulfils the second subquery, the one which has the highest rank contains the following operations:

- FromCoordinatesToCity: Given the geographical coordinates, this operation returns the name of the city where are allocated the coordinates of certain point of interest.
- CoordinatesFromAddress: Given an address, this operation returns its geographical coordinates.

- GetDirection: This operation provides instructions on how to reach a destination. This operation receives two parameters, the coordinates of the starting point, and the coordinates of the destination.

The module COS takes the operations of both services and compose them. The execution of the resulting composite service fulfils both Alice's needs (i.e., booking a table in the finest restaurant of the city, and knowing the direction to go there).

On that night, while she is enjoying a delightful dinner in *Le Meurice* restaurant, Alice is wondering about the weather in the next day. She needs this information to decide whether she will go to *Louvre museum* or *Euro Disney*. One more time, she uses the system and submits the following query: I want to buy a ticket for Euro Disney tomorrow if the weather forecast is sunny, otherwise, buy a ticket for Louvre museum. At this time, Alice's profile is still the same, however, her new spatial context information is as follows: coordinates = "48.8651° N, 2.3280° E" and date = "1/06/2014".

Similar to the previous query, this one contains three requirements, therefore the UIQM module splits the query in three subqueries. The first subquery is to buy a ticket for Euro Disney tomorrow. The second subquery is the weather forecast is sunny. The last subquery is to buy a ticket for Louvre museum. All three subqueries are sent to the module DS, thereby it sends three lists of services to the module COS. From the list of candidate services that may fulfil the first subquery, the one which has the highest rank contains the following operation: BuyTickets4EuroDisney, this operation receives as parameters the name of the customer, the number of required tickets, information of a credit card, etc. As a result, the operation returns a confirmation whether the transaction has been successfully finished or not.

In another ranked list of candidate services that may fulfil the second subquery, the one which has the highest rank contains the following operation: GetWeatherForecast, this operation returns the weather for a given city of a certain country, and for a given date at a given time.

In the ranked list of candidate services that may fulfil the last subquery, the one which has the highest rank contains the following operation: BuyTickets4LouvreMuseum, this operation receives similar operation as the one to by tickets for *Euro Disney*, besides, the result of this operation is the same.

In the same way as before, the COS module composes all operations of previous services. The execution of the resulting composite service fulfils Alice's requirements regarding her condition. Once the process is finished, Alice is ready for the tourist activities that she has planned to do in the next day, in this scenario she has bought a ticket for Louvre museum because of the next day shall be raining.

With the above aforementioned system, Alice is able to consume services provided in the Web, from her mobile devices. Furthermore, service provides do not need to produce front-end applications, which serve as interfaces to access their services. However, the back-end process is complex since one service cannot fulfill all user's need at one time. This thesis addresses the problem of automated composing service operations and executing a result of the service composition, so that the result fulfills specific users' requirements.

1.3 Problem Definition

To pursue the development of a Service-Oriented system, several characteristic such as software layers can be used to control the system implementation and provide a logical application structure [Erl05].

Choreography layer defines a skeleton of communication among all business participants. The business participants, for example, can be suppliers, merchants and consumers. A fact behind is that all business participants desire to achieve certain goals together. Thus, in service-oriented system, the interactions are derived from all participants. They agree on the behavior in terms of messages that are exchanged among them in a business process.

Orchestration layer refers to a business process that interacts with web services. The interactions are derived from the participants in the choreography that is required to realize and implement the business process. The construction of the business process is depended on its business logic and execution order of the interactions. Additional, the process is managed from the perspective of one of the business participants.

Service layer consists of all atomic and composite services that are available for the upper layers to integrate them in compositions or use them as part of the choreography description.

Execution layer comprises of all aspects related to the SOA capabilities such as

publish-find-bind, the dynamic invocation of services and execution of composite services and business processes.

There are two approaches for developing such a system. The first approach is called a top-down approach. With this method, the system development starts at Choreography layer to define a common agreement among the participants. Later, the choreographies are transformed into orchestrations to be executed by a platform engine. Contrary, a bottom-up approach produces orchestrations from composing existing services to reach user's requirements. The user's requirements can be in form of either user's goal behavior such as composite service specification or user's goal in propositional.

Our service composition and execution component in E-Tourism system (see in Section 1.1) is suitable for the bottom-up approach. Since the component has to compose operations, offered by candidates services, to fulfill users' needs. Developing such service-oriented systems with the bottom-up approach have been discussed for many decades ago. Many previous works present a system framework covering the above mentioned layers. However, only fews of them have carried out the system implementation from choreography layer to execution layer; they have focused on logical composition in orchestration layer.

1.4 Research questions

Since user query is often complex, an operation offered by a web service is unlikely to fulfill the user query. For example, I want to book a table for 2 people at the finest restaurant query needs at least 2 operations such as findRestaurant and bookTable operations combining to answer the query. Therefore, we need a system performing the composition of service operations. So far, we have discussed in the previous section that such a system is suitable for the bottom-up approach. However, we also want the system to support on the fly requests from users. In other words, the system should be able to generate a custom application as quickly as possible. To achieve this goal, we divide works happened at design time from those which happened at implementation time. At design time, an agent with provided knowledge takes the role of a software designer to design a workflow of service operations. This is called automated service composition, which minimizes time consuming and complexity of tasks such as composition

requirement analysis. While at the implementation time, we realize that instead of using the structure or object-oriented programming, an application may be synthesized from business process languages such as BPEL[6] and BPMN[7]. Since operations and controls among them in the service composition domain are similar to tasks and gateways in the business process domain.

The aforementioned issues raise the need for a set of methods and a framework to effectively develop *Automated service composition and execution system*. This dissertation is guided by the following two main research questions including a subquestions to further structure the main question.

Q1: How to compose services so that the final result of composite service satisfies users' needs?

- Which methods provide a flexible and effective way to specify the behavior of a service composition?
- Which developing system approach (top-down or bottom-up approaches) is suitable for our service composition system?
- Which techniques can tackle automated service composition?

Q2: How to execute the result of composite service?

- What kind of runtime support mechanisms are required to address service composition and execution?
- How to transform the resulting composite service into an orchestration model which includes control and data flows?
- How to acquire parameters from users, which are used in data flow model?
- How to deploy orchestration instance, which is later executed by orchestration engine?

[6]Business Process Execution Language (BPEL) is an OASIS standard executable language for specifying activities within business processes with web services.

[7]Business Process Model Notation (BPMN) is a graphical representation for determining business processes in a business process model.

1.5 Dissertation aims

According to the research questions, we summarize in this section all contributions of this dissertation. They cover a research challenge and integration architecture to give the big picture description of this dissertation. The architecture is depicted in Figure 1.2 and comprises four different layers. We annotated the architecture to associate with the specific contributions: (1) the multi-layer model of the service composite and execution system framework, (2) the abstract composition in logical layer and (3) the business process generation in composition platform layer. Below, we comment the contributions:

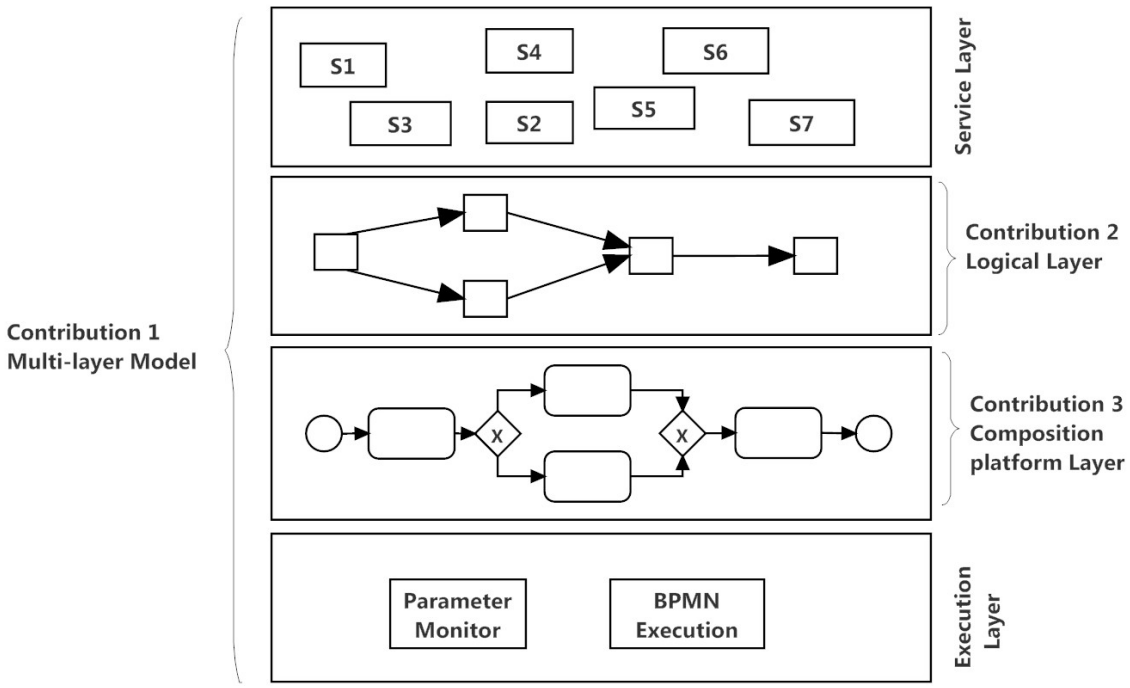


FIGURE 1.2: Multi-layer for service composition and execution system

Multi-layer model. The first contribution of this dissertation is a service composition model comprising of multiple layers. From the top layer, relevant services to fulfill users' needs can be accessed from service providers in the *Service layer*. In this PhD, we consider web services, which explain operation(s) with an invocation interface to call these operations. For example, restaurant service offering two operations[8] searchRestaurant:city,type $\mapsto$ restaurantName and bookTable:restaurantName,guestName $\mapsto$ confirmationBooking. The next is *Logical*

[8]Parameters after colon refer to operation inputs while after arrow are for operation outputs.

layer, which contains a logical composition of operations. The operations from the service layers are assembled into a workflow against user's goals.

For example, the sequence of calls to searchRestaurant then bookTable is satisfied against confirmationBooking. However, the generated workflow is not in an executable form, this is why we perform the transformation of the workflow into an executable business process in the *Composition platform layer*. The last layer is the *Execution layer*, which handles runtime activities such as binding services, acquiring missing parameters and enacting business process engine.

Logical Layer. To support the development of the service composition component, we adopt, in the logical layer, a bottom-up modeling approach and automated technique for service composition. The planning technique taken from the artificial intelligent domain is chosen to design an agent for the orchestration of service operations. Generally speaking, given starting state(s), goal state(s) and a set of actions, the agent finds a sequence of actions (or abstract plan) that get the agent from the start(s) to the goal state(s). We propose constraints programming system being our planning agent to solve service composition problems. Our agent is capable to generate the abstract plan, which has sequencing, conditioning and parallelizing controls among abstract service operations. Further details of the logical layer will be given in Chapter 3.

Composition platform Layer. Since we want to minimize time consuming for the implementation and execution of service composition, we choose BPMN as a target execution platform. However, there is a gap between abstract plan generated from the previous layer and executable BPMN platform. With the aforementioned problem, this dissertation contributes a novel automated model checking approach for correctness of user requirements and BPMN specification. In this layer, a proposed algorithm to transform the abstract plan into BPMN model. Furthermore, the resulting orchestrations are valid against BPMN specification rules. The Composition platform generation are detailed in Chapter 4.

Execution Layer. The BPMN instant generation is involved in this layer. We implement BPMN model to read the valid semantic BPMN from the previous layer and we use open sourced APIs of business process modeling to generate BPMN instant source code. At the same time, all input parameters are required for executing the BPMN instant. These issues on the parameter acquiring and the BPMN implementation and execution are discussed in Chapter 5.

Finally, we have implemented a prototype to realize our service composition and

execution approach as a proof of concept. The prototype illustrates a composition as a service based on logic language, allowing automated composition generation, executable platform transformation and execution of composite services. In addition, we have setup scenarios from E-tourism domain to perform an evaluation of the prototype. More detail of the implementation is shown in Chapter 5.

1.6 Dissertation outline

This dissertation is organized as follows:

- Chapter 2 is divided in two sections. Section 1 comprises essential information as the background of the service composition context. We start with the question of why a cooperation among participants is important. The concept of Business-to-business transaction is introduced. Examples of B2B system are mentioned. Afterward, we come across the technical information of how to invent such system. Service-oriented computing, web service and semantic web service are discussed. Section 2 details the related work on service composition. We classify the work into three levels (framework level, logical level and business process generation level) as part of the contributions.

- Chapter 3 presents an overview of our logical layer to automated workflow of Abstract service composition system. The overall requirements of the system are elicited. Since we see service composition problem as planning problem, we separate the system into two processes: Transformer process and Planner process. The Transformer changes the user's context and query into planning problem and transforms a list of service operations into a set of possible actions. The Planner shall try to reason the updated state of the given actions to obtain a workflow of actions, which satisfies the planning problem. We demonstrate our algorithms of the proposed system with one scenario from the motivating examples in Section 1.2.

- Chapter 4 presents our composition platform layer to validate semantic BPMN model of the composite platform generation system. The objective of the system is to analyze and check the properties of derived BPMN model from the workflow of abstract service composition system. Therefore,

a model representing BPMN process in Prolog language is proposed in this chapter. Two processes of BPMN Transformer and BPMN Validation respectively are presented and detailed. We demonstrate our algorithms of the proposed system with the same scenario used in Chapter 3.

- Chapter 5 presents implementation of our system for a proof of concept. It covers the abstract service composition system in Chapter 3, the composite platform generation system in Chapter 4 and the parameter monitor and BPMN execution components happened in the execution layer. We present meta-model used to predefined objects of our proposed system. Finally, we show the experiments and results of our setup scenarios from Chapter 1.

- Chapter 6 concludes the research and offers future research recommendations.

Chapter 2

Background and State of the Art

Contents

Abstract. This Chapter describes background knowledge, concepts relevant to B2B interactions, Service-Oriented Architecture (SOA), web services and service composition approaches. In this Chapter, we present idea and existing research achievement in automated service composition and execution framework.

Research on service composition system tackles the problem of developing extensible and stable applications that satisfy some user's needs. It incorporates works

from several disciplines such as analysis and inference, modeling, transforming, aggregating, validating, testing as well as service composition. This chapter describes background and related work in four main research categories:*B2B interactions, Service-Oriented Computing, Web service, Semantic Web service and Service Composition*. Section 2.1 discusses the concept of B2B interactions which are used to describe how business partners exchange all products and/or service transfers. In Section 2.2, we detail Service-Oriented Architecture (SOA) which is the grounded in the idea of service composition, web services, which is one of SOA technologies, how to construct and access web services (architecture and technologies related). Semantic web services which significantly facilitate automated service discovery, service composition and service execution are also discussed. Finally in Section 2.3, we introduce service composition, which is the core of our work. We start with the life cycle of service composition illustrated to realize the process of constructing software or application from service components. Then a discussion between manual service composition and automated service composition is raised. However, our work is dominated by automated service composition approach (see Section 1.4). We explore Artificial Intelligent (AI) planning technique to deal with the automated service composition problem. The chapter concludes by highlighting the shortcoming of the existing approaches based on service composition.

2.1 B2B Interactions

The growth of Electronic commerce (or called E-commerce) had been driven along with the growth of the Web [Dog98, SBe00]. The first generation of Web-based E-commerce was Business-to-Customer (B2C) Applications. These applications create online channels between businesses and customers to do activities of businesses in selling products and/or services. For example, someone purchasing some books from amazon.co.uk or ordered CDs/DVDs from play.com. However, in order to have these B2C transactions run smoothly and efficiently, the businesses might need to perform some establishing and supporting actions such as procurement, human resources, marketing, sales, supply chain and manufacturing. For example, when receiving orders from customers, Amazon company contacts and purchases

the ordered books from publisher companies. Such actions, most of the time, occur in Business-to-Business (B2B) interactions. The B2B E-commerce interactions describe all transactions of products and/or services between businesses.

Since network brings freely communication between enterprises via the web to achieve business integration of complementary establishment and beneficial cooperation. Buyers and sellers on B2B internet-based transactions reduce a lot of resources and time in order to complete the entire business process from business establishment till customer service[1].

However, some challenges have appeared on B2B interactions. Consider guest house providing customers with a variety of room types such as single, double and family. To run its business, the guest house sets up four systems (Payroll, Booking, Time-stamp and Billing systems). Each sub-system has its own user interfaces and databases. The payroll system is for calculating salaries for the guest house's working staffs. The booking system deals with room reservations for customers. The time-stamp system monitors and records time when the staff members punch in/out the clock. And the billing system is for issuing customers' quotations and receipts. So far, customers fill check-in, check-out dates and number of guests on the guest house's web site. The system will provide a list of available rooms including room description, photos and prices. After selecting the desired rooms, customers finish up the payment and receive back the room receipts.

The challenge in such setting this business would be to enable the system to interoperate to each others. For instance in guest house example, the generation of customers' bills retrieves the number of nights the customers stayed in the guest house from the booking system and calculation of staffs' wages depends on their working hours recorded by the time-stamp system.

However, same problems of connectivity among applications might occur. Different systems may have incompatible protocol connectivities. For example, billing application is DCOM[2] computer program written in Visual Basic language while booking room application is written in Java programming language which uses Java RMI[3].

[1] http://faqb2b.blog.com/what-is-b2b/

[2] Distributed Component Object Model (DCOM) is a platform-independent, distributed, object-oriented system for creating binary software components that can interact.

[3] Java Remote Method Invocation (Java RMI) enables the programmer to create distributed Java technology-based to Java technology-based applications possibly on different hosts.

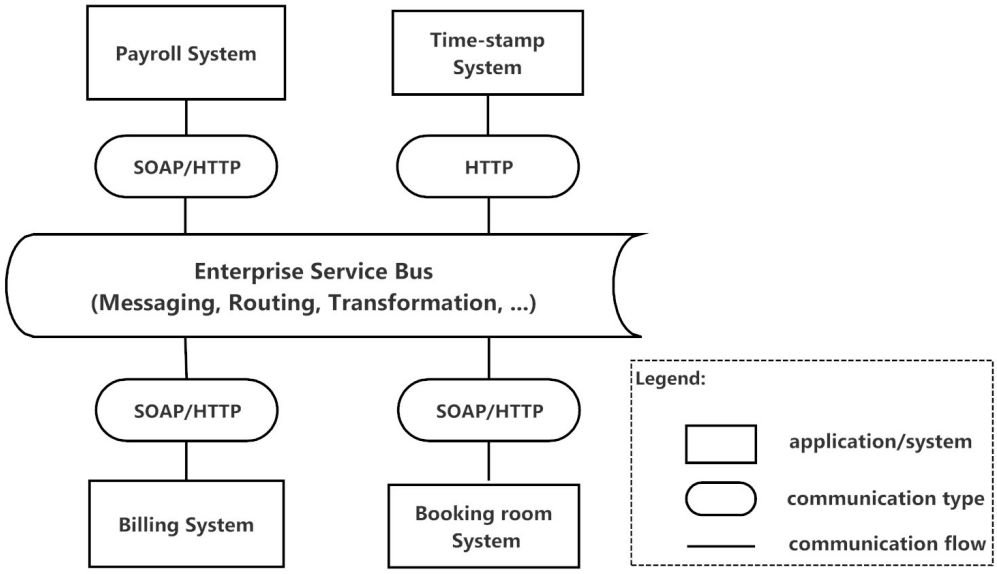


FIGURE 2.1: The guest house system relying on ESB [7]

To deal with the above mentioned interoperability problem, Enterprise Service Buses (ESB) approach was introduced to handle the communication among applications using message over a network [ESB10]. Those messages could use several supports: from the heavy SOAP[4] specification to the REST[5] lightweight principle, even RPC[6] is a possible support for messaging in an ESB. In Figure 2.1, we illustrate the guest house system with an integration of the Payroll sub-system, the Booking sub-system, the Time-stamp sub-system and Billing sub-system under ESB architecture.

2.1.1 Framework Layers

In spite of the ESB offered the solution of an interconnection among diverse applications over the network, it deals with only communication level. To run B2B application successfully, business partners need to work together on interactions layers. Interactions in B2B application occur in three layers: *communication*, *content* and *business process* layers [MBB+03]:

[4]Simple Object Access protocol (SOAP) is a protocol specification for exchanging structured information in the implementation of web services in computer networks.

[5]Representational State Transfer (REST) is a software architecture style consisting of guidelines and best practices for creating scalable web services.

[6]Remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space.

[7]inspired from http://www.fiorano.com/products/ESB-enterprise-service-bus/Fiorano-ESB-enterprise-service-bus.php

- **Communication layer** offers standard protocols for exchanging messages among remotely located partners. It is possible that partners use distinct proprietary communication protocols. In this case, gateways should be used to translate messages between heterogeneous protocols.

- **Content layer** provides languages and models to describe and organize information in exchanging messages to be standard and understood by a machine. The content interactions require that the involved system understand the semantics of content and syntax of types in documents. XML for instance, particularly XML Schema, can handle communication formatting of a given message.

- **Business process layer:** A business process (or a workflow) is a structured and measured set of activities designed to produce a specified output for a particular customer or market [Dav93]. However in B2B interaction domain, the designed activities may outsource the services from other business process. The ambiguities such as what an exchanged message may mean, what actions are allowed and what response are expected may occur. Thus the semantic of interactions among all business partners must be defined well. This layer addresses the conversational interactions (i.e, joint business process) among services.

This thesis follows the layers (Communication, Content, and Business process layers) in B2B interaction framework [MBB+03] to pursue the objectives of an automated service composition and execution framework as stated in chapter 1. The rest of this chapter continues with Service-oriented computing, Web service and Semantic web service in Section 2.2 covering aspects of the communication and content layers in B2B interactions. While the service composition is introduced in Section 2.3 to express techniques and technologies used to build the control and data flows in business process layer.

2.2 Service-Oriented Computing

The principle idea behind service computing is to create a software application by reusing existing other software applications. The program can inter-operate with existing software components to obtain the objective goal. However, in order

to achieve this goal, inter-operating components rules shall be strictly complied. These rules conform to Service-Oriented Computing (SOC).

Service-Oriented Computing (SOC) is the computing paradigm that utilizes services as fundamental elements for developing applications/solutions [PvdH07]. The size of application development is diverse from services within an computer program to service applications inter-acting across enterprises.

SOC addresses these problems by providing the abstractions and tools to model the information and relate the models, construct processes over the systems, assert and guarantee transactional properties, add flexible decision-support, and relate the functioning of the component software systems to the organizations that they represent [HS05].

To build the software application, SOC relies on the Service Oriented Architecture (SOA), which is a way of reorganizing software application and infrastructure in to a set of interacting services [PvdH07]. A definition of SOA is given below.

> *SOA is an architectural paradigm for components of a system and interactions or patterns between them such that a component offers a service that queue in a state a readiness and other components may invoke the service in compliance with a service contract.* [Nic05]

What is the definition of service? *"Services are self-describing, open components that support rapid, low-cost composition of distributed applications"* in another definition by the same author [PvdH07]. Service is a business function, which is implemented in a software format and supplied with a widely intelligible formal documented interface [PvdH07].

General speaking, SOA is an architecture that governs a set of services exposed by service provider to achieve the desired end results for a service consumer. Consequently, to satisfy these requirements above, services should be subjected to the following major requirements [PvdH07]:

- **Technology neutral.** Service must not rely on or be bounded to concrete implementation technologies and standards utilized at both client and service sides. Instead services have to be operated by mechanisms (protocols, descriptions and discovery mechanisms) that comply with widely recognized standards.

- **Loosely coupled.** No knowledge regarding any internal structures both at client and service sides is required for implementing an application that calls service operations.
- **Support location transparency.** A service should have its description and location information stored in a repository and be accessible by a variety of clients that can locate and invoke it irrespectively of its actual location.

SOA can be implemented using a wide range of technologies such as J2EE[8], .NET[9] and many others. However, the most popular and complete implementation of SOA is based on Web Services[10]. We give background of web service and its terminology, standards and specification below.

2.2.1 Web Service

Web services were introduced at the beginning of the 2000s. At that era, all predictions agreed that B2B E-commerce will be booming in worth billions of dollar in new investment [BMB+00]. Since Cheap connectivity and ease of advertising of data and services on the Web created tremendous opportunities for organizations of any size to diversify their customer-base and become truly global [BEB98]. Therefore, the needs of business protocol on the web (or web services) had been originated. In other words, web services brings a standard for business partners communication over a network.

Before the Web service technology, the Business to Business (B2B) applies a conventional middle-ware platform to do commercial transactions among business partners [ACKM04]. The transactions are fully automated between companies. For instance, a restaurant owner uses the conventional middle-ware platform to offer restaurant reservation for consumers. Therefore, instead of manually process a table booking, customer fills a request form and places the booking electronically directly with the restaurant provider. At restaurant provider side, the booking can then be immediately processed and a confirmation sent to the customer. This approach facilitates the business partners and is less prone to human errors that might occur in business workflow.

[8] http://java.sun.com/j2ee/
[9] www.microsoft.com/net/
[10] http://www.w3.org/2002/ws/

However, the conventional middle-ware platform has limitation of the integration of several autonomous and heterogeneous systems. The companies lack of standardization across concrete middle-ware platforms. There are some issues regarding the centralized middle-ware or third party trustworthy. Every company does want to keep confidential its business transaction . The solution for this problem is a point-to-point integration across companies. This means that all companies need to have a co-agreement on middle-ware protocol and infrastructure. However, this solution will be more costly if there are different partners and each partner require the use of a different middle-ware platform.

The described limitations of the middle-ware platforms are solved by an approach of application integration technology using web services. Since each web service has an interface that describes a collection of operations accessible on HTTP[11] protocol internet via standardized XML[12] messaging, many of business partners are interested in and agree to adopt web services as an application integration technology.

We quote below definition of Web service given by the World Wide Web consortium (W3C).

> *"A web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL[13]). Other systems interact with the Web service in a manner prescribed by its description using SOAP[14]-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."*

Thus with web service technology, each business partner can expose the existing functions in any technology and platform on the network. Later, other applications can access the exposed functions. In other words, web services bring the application interoperability. In addition, the cost for implementing Web service is much less than the mentioned middle-ware platform. One company can implement Web service using an existing low cost internet. Since web services use SOAP over HTTP protocol for the communication.

[11] http://www.w3.org/Arena/webworld/httpwgcharter.html
[12] http://www.w3.org/TR/REC-xml/
[13] http://www.w3.org/TR/wsdl
[14] http://www.w3.org/TR/soap/

2.2.1.1 Architecture

The following subsections give more details on how SOA works with web service.

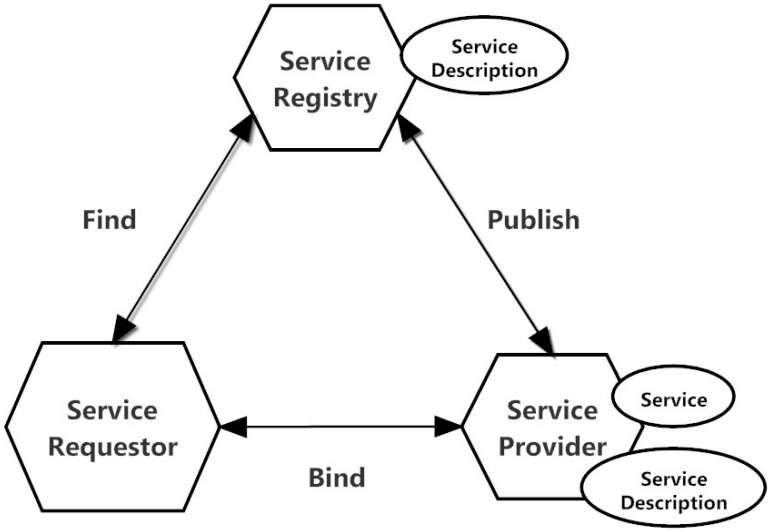


FIGURE 2.2: Web service architecture

Web service architecture describes the interactions between three roles which are the service requestor, the service provider and the service registry. The service requestor looks for the targeted services from the advertised services in the service registry. If the targeted services are found, the service requestor will invoke and bind service port types from service provider with its operations. The web service architecture model shown in Figure 2.2 illustrates the interaction between components, component's operations and artifacts.

Roles in web service architecture

- **Service Requestor** is any application or service which wants to use particular services published by service provider. After locating desired service in service registry offering various operations, the service requestor binds the services to the service providers. So that the service requestor is able to invoke the web services.
- **Service Provider** is a service owner which wants to publish/expose its services accessible on the internet towards other applications. The service provider hosts both implemented service and its web service description (WSDL) document. The WSDL document (or simply contract of service)

describes service access information such as interface, data types, binding information and network location.

- **Service Registry** is a searchable respository of a set of advertised service descriptions. The service registry allows service providers to publish their service descriptions and then classify the service descriptions according to searching criteria such as type of services and type of businesses.

2.2.1.2 Technology

As described in web service architecture, one of the artifacts which plays an important role is the Web Service Description Language (WSDL). This section explains the most significant details of WSDL.

W3C gives the definition of WSDL *"that is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint."* [CCMW01]

A WSDL document defines services as a collection of network endpoints, or ports. Each port define a abstract set of operations, each operation specifies data exchanged between input messages and output messages. The data type of each message can be either simple type or complex type. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service.

In Figure 2.3 is depicted an interface of WeatherForecast service. The WeatherForecast service provides information of weather forecast for given specific date and city.

FIGURE 2.3: WeatherForecast service interface

To detail the WeatherForecast service, we show in Listing 2.1 its fragment example code (line 1-58) of its WSDL document. Under the element *portType* WeatherforecastPortType (line 35), the operation GetWeatherforecast is defined. The operation

GetWeatherforecast has two inputs and one output under *message* GetWeather-RequestInput (line 27) and *message* GetWeatherRequestOutput respectively. The input message refers to *element* WeatherRequest (line 13) which is correspond to *element* date typed Date (line 15) and *element* cityName typed string (line 16). The output message refers to *element* WeatherForecast (line 19) which is correspond to *element* WeatherforecastInfo typed String (line 21). To invoke this service Weatherforecast, the element *binding* WeatherforecastSoapBinding (line 42) defines protocol, data format for operation and messages in Weatherforecast-PortType. The element *port* WeatherforecastPort (line 53) is defined by element *service* WeatherforecastService (line 54) to specify an communication address "http://example.com/weatherforecast".

```
1  <?xml version="1.0"?>
2  <definitions name="WeatherForecast"
3
4  targetNamespace="http://example.com/weatherforecast.wsdl"
5  xmlns:tns="http://example.com/weatherforecast.wsdl"
6  xmlns:xsd1="http://example.com/weatherforecast.xsd"
7  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
8  xmlns="http://schemas.xmlsoap.org/wsdl/">
9
10 <types>
11 <schema targetNamespace="http://example.com/weatherforecast.xsd"
12 xmlns="http://www.w3.org/2000/10/XMLSchema">
13         <element name="WeatherRequest">
14         <complexType>
15                 <element name="date" type="date"/>
16                 <element name="cityName" type="string"/>
17         </complexType>
18         </element>
19         <element name="WeatherForecast">
20         <complexType>
21                 <element name="weatherforecastInfo" type="string"/>
22         </complexType>
23         </element>
24 </schema>
25 </types>
26
27 <message name="GetWeatherRequestInput">
28 <part name="body" element="xsd1:WeatherRequest"/>
29 </message>
30
31 <message name="GetWeatherForecastOutput">
32 <part name="body" element="xsd1:WeatherForecast"/>
33 </message>
34
35 <portType name="WeatherForecastPortType">
36         <operation name="GetWeatherForecast">
37                 <input message="tns:GetWeatherForecastInput"/>
```

```
38                  <output message="tns:GetWeatherForecastOutput"/>
39          </operation>
40  </portType>
41
42  <binding name="WeatherForecastSoapBinding" type="tns:WeatherForecastPortType">
43  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
44          <operation name="GetWeatherForecast">
45          <soap:operation soapAction="http://example.com/GetWeatherForecast"/>
46          <input><soap:body use="literal"/></input>
47          <output><soap:body use="literal"/></output>
48          </operation>
49  </binding>
50
51  <service name="WeatherForecastService">
52          <documentation>Weather Forecast service</documentation>
53          <port name="WeatherForecastPort" binding="tns:WeatherForecastBinding">
54                  <soap:address location="http://example.com/weatherforecast"/>
55          </port>
56  </service>
57
58  </definitions>
```

LISTING 2.1: WeatherForecastRequestService in WSDL document

Thus service providers advertise their web services by registering the WSDL documents at some service registry. Service requestors can then process service discovery for the particular web services at the same service registry and later bind service endpoints to their applications. Under B2B interaction techniques with Web services, the involved partners benefit advantages including *coupling among partners*, *heterogeneity* and *autonomy*. Web services allow interactions at the communication layer by using SOAP as a messaging protocol. The adoption of an XML-based messaging over well-established protocol (e.g., HTTP, SMTP[15], and FTP[16]) enables loosely coupled communication among heterogeneous systems [MBB+03]. The heterogeneous applications may be wrapped and exposed as web services. In term of autonomy, Web services are accessible through published interfaces. This enables the partners have more local control over implementation and operation of services, and flexibility to change their processes without affecting each other [MBB+03].

Nonetheless, WSDL documents lack providing data format, data models and languages which help the involved systems to understand the semantics of content and types of business document. From the guest house system, if the billing system receives a message form the booking room system, the billing system must

[15] http://tools.ietf.org/html/rfc5321
[16] http://tools.ietf.org/html/rfc959

determine whether the content in the message represent a request for a receipt or a room quotation. In addition, the so far processes such as service discovery and service composition are done manually which might be error prone and time consuming. The semantic approach for web services is given in the next section, overcome some of these above mentioned problems.

2.2.2 Semantic Web Service

There are increased web accessible programs, databases, and sensors primarily in B2B and e-commerce applications [MSZ01]. Since one web service cannot complete a business process, the involved applications often realize automation of web service interoperation.

In this dissertation, we concern an automation of web service interoperation for two reasons. The first reason is to facilitate the Discovery System (DS) searching for composable service operations subjective to user's requirements expressed in free text queries [CC14]. The second reason is to apply planning technique for the orchestration of returned service operations; each operation has an interface with input and output entities from a semantic data model.

The realization of the Semantic Web comes from the development of new content markup languages, such as OIL [FHH+01], DAML+OIL[17] and OWL-S[18]. These languages have a well-defined semantics and enable the markup and manipulation of complex taxonomic and logical relations between entities on the web. A fundamental component of the semantic web will be the markup of web services to make them machine-interpretable, use-apparent, and computer-ready. So computer agents can reason about web services to perform automated web service discovery, web service composition and interoperation and then web service execution [MSZ01]. In this section, we present and compare two dominant approaches for Semantic web services: Semantic Annotations for WSDL and XML Schema (SAWSDL) and Semantic Markup for Web Services (OWL-S).

SAWSDL is a W3C Recommendation which defines it as mechanisms using which semantic annotations can be added to WSDL components [FL07]. SAWSDL standard deals with problems regarding to data heterogeneity in WSDL

[17] http://www.daml.org/2001/03/daml+oil-index
[18] http://www.w3.org/Submission/OWL-S/

2.0 [CMRW07]. Two WSDL documents can have similar descriptions while meaning totally different things, or they can have very different descriptions yet similar meaning. Therefore, SAWDL provides mechanisms by which concepts from the semantic models that are defined either in between or outside the WSDL document can be referenced from within WSDL components as annotations. These semantics when expressed in formal languages can help disambiguate the description of web services during the automated discovery and composition of the web services [FL07].

Consider the Listing 2.2 fragment code modified from the previous WeatherForecastRequest WSDL example (introduced in Listing 2.1). This service offers the requesters to get the weather forecast for a given date and city (see Figure 2.3, Page 24). To realize a semantic annotation for WSDL, the service provides an annotation sawdl:modelReference of a semantic model named SampleOntology points to the matching attributes defined in WSDL. For example, element named date refers to the vocabulary date (line 13), element named cityName refers to the vocabulary city (line 15) and element named WeatherForecastInfo refers to the vocabulary weather (line 21). Thus if the service request WSDL is annotated using the same semantic model, a semantic engine could use this information to match the two web services. Without the semantic annotations, the matching engine may not have sufficient information to identify them as related terms unless explicitly specified.

```
1  <wsdl:description
2    targetNamespace="http://example.com/wsdl/weatherforecastRequestSevice/"
3    xmlns="http://example.com/wsdl/weatherforecastRequestSevice/"
4    xmlns:wsdl="http://www.w3.org/ns/wsdl"
5    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6    xmlns:sawsdl="http://www.w3.org/ns/sawsdl">
7
8    <wsdl:types>
9      <xsd:schema targetNamespace="http://example.com/wsdl/weatherforecastRequestSevice">
10     <xsd:element name="WeatherRequest">
11       <xsd:complexType>
12           <xsd:element name="date" type="xsd:date"
13             sawsdl:modelReference="http://example.com/ontologies/SampleOntology#date"/>
14                 <xsd:element name="cityName" type="xsd:string"
15             sawsdl:modelReference="http://example.com/ontologies/SampleOntology#city"/>
16       </xsd:complexType>
17     </xsd:element>
18         <xsd:element name="WeatherForecast">
19       <xsd:complexType>
20           <xsd:element name="WeatherForecastInfo" type="xsd:string"
21             sawsdl:modelReference="http://example.com/ontologies/SampleOntology#weather"/>
```

```
22        </xsd:complexType>
23      </xsd:element>
24      </xsd:schema>
25    </wsdl:types>
26
27     <wsdl:interface name="WeatherForecastRequestService">
28        <wsdl:operation name="WeatherForecastRequestOperation"
29                  pattern="http://www.w3.org/ns/wsdl/in-out">
30        <wsdl:input element="WeatherForecasteRequestServiceRequest"/>
31        <wsdl:output element="WeatherForecastRequestServiceResponse"/>
32      </wsdl:operation>
33    </wsdl:interface>
34
35
36 </wsdl:description>
```

LISTING 2.2: WeatherForecastRequest SAWSDL Service

OWL-S is an OWL based ontology for describing Semantic Web Services. It will enable users and software agents to automatically discover, invoke, compose and monitor web resources offering services under specified constraints [MBH+04]. To facilitate the OWL-S capacities above mentioned, OWL-S organizes the service structure into three parts which are service profile, process model and service grounding. To give more details of OWL-S structure, the same WeatherForcastRequest service, but formatted in OWL-S form is illustrated in Listing 2.3. The service profile part (line 26-39) is used to describe what the service does, which includes the information such as the service name and description, quality of service, publisher and contact information. The process part (line 40-56) describes the process type and its elements (a set of inputs, outputs, preconditions, effects of the service execution) inside the process. In the example code, the WeatherForcastRequest service defines its process type as AtomicProcess that is binded to a single operation GetWeatherForecastPrice. Furthermore, OWL-S services offer two more process types to support complex business processes. The former (called CompositeProcess) is a process that requires multiple actions from other process, in which directed by one of control constructs such as sequence, iterate, choice and if-then-else. The latter (called SimpleProcess) is a process that provides an abstraction mechanism that offers multiple views of the same process [MBH+04]. Finally, the service grounding (line 65-113) specifies the interaction information with the service such as communication protocols, message formats and port number.

```
1 <?xml version="1.0" encoding="WINDOWS-1252"?>
```

```
<rdf:RDF  xmlns:owl       = "http://www.w3.org/2002/07/owl#"
xmlns:rdfs      = "http://www.w3.org/2000/01/rdf-schema#"
xmlns:rdf       = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:service   = "http://www.daml.org/services/owl-s/1.1/Service.owl#"
xmlns:process   = "http://www.daml.org/services/owl-s/1.1/Process.owl#"
xmlns:profile    = "http://www.daml.org/services/owl-s/1.1/Profile.owl#"
xmlns:grounding = "http://www.daml.org/services/owl-s/1.1/Grounding.owl#"

xml:base        = "http://127.0.0.1/services/1.1/weather_forecast_service.owls">

<owl:Ontology rdf:about="">
        <owl:imports rdf:resource="http://127.0.0.1/ontology/Service.owl" />
        <owl:imports rdf:resource="http://127.0.0.1/ontology/Process.owl" />
        <owl:imports rdf:resource="http://127.0.0.1/ontology/Profile.owl" />
        <owl:imports rdf:resource="http://127.0.0.1/ontology/Grounding.owl" />
        <owl:imports rdf:resource="http://127.0.0.1/ontology/weather.owl" />
</owl:Ontology>

<service:Service rdf:ID="WEATHER_FORECAST_SERVICE">
        <service:presents rdf:resource="#WEATHER_FORECAST_PROFILE"/>
        <service:describedBy rdf:resource="#WEATHER_FORECAST_PROCESS"/>
        <service:supports rdf:resource="#WEATHER_FORECAST_GROUNDING"/>
</service:Service>

<profile:Profile rdf:ID="WEATHER_FORECAST_PROFILE">
        <service:isPresentedBy rdf:resource="#WEATHER_FORECAST_SERVICE"/>
        <profile:serviceName xml:lang="en">
                WEATHER_FORECAST Service
        </profile:serviceName>
        <profile:textDescription xml:lang="en">
        This service returns the information of weather forecast given by date and city name.
        </profile:textDescription>
        <profile:hasInput  rdf:resource="#_DATE"/>
        <profile:hasInput  rdf:resource="#_CITY"/>
        <profile:hasOutput rdf:resource="#_WEATHERINFO"/>

        <profile:has_process rdf:resource="WEATHER_FORECAST_PROCESS" />
</profile:Profile>

<process:AtomicProcess rdf:ID="WEATHER_FORECAST_PROCESS">
        <service:describes rdf:resource="#WEATHER_FORECAST_SERVICE"/>
        <profile:hasInput  rdf:resource="#_DATE"/>
        <profile:hasInput  rdf:resource="#_CITY"/>
        <profile:hasOutput rdf:resource="#_WEATHERINFO"/>
</process:AtomicProcess>

<process:Input rdf:ID="_DATE">
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        http://127.0.0.1/ontology/weather.owl#date</process:parameterType>
        <rdfs:label></rdfs:label>
</process:Input>
<process:Input rdf:ID="_CITY">
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        http://127.0.0.1/ontology/weather.owl#city</process:parameterType>
        <rdfs:label></rdfs:label>
```

```
</process:Input>

<process:Output  rdf:ID="_WEATHERINFO">
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        http://127.0.0.1/ontology/weather.owl#weather</process:parameterType>
        <rdfs:label></rdfs:label>
</process:Output>

<grounding:WsdlGrounding rdf:ID="WEATHER_FORECAST_GROUNDING">
<service:supportedBy rdf:resource="#WEATHER_FORECAST_SERVICE"/>
<grounding:hasAtomicProcessGrounding>
    <grounding:WsdlAtomicProcessGrounding rdf:ID="WEATHER_FORECAST_AtomicProcessGrounding"/>
    </grounding:hasAtomicProcessGrounding>
</grounding:WsdlGrounding>

<grounding:WsdlAtomicProcessGrounding rdf:about="#WEATHER_FORECAST_AtomicProcessGrounding">
        <grounding:wsdlDocument rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://127.0.0.1/wsdl/WeatherForecast.wsdl</grounding:wsdlDocument>
    <grounding:owlsProcess rdf:resource="#WEATHER_FORECAST_PROCESS"/>
    <grounding:wsdlOperation>
        <grounding:WsdlOperationRef>
        <grounding:operation rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        http://127.0.0.1/wsdl/WeatherForecast/GetWeatherForecastPrice</grounding:operation>
        <grounding:portType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        http://127.0.0.1/wsdl/WeatherForecast/WeatherForecastPortType</grounding:portType>
        </grounding:WsdlOperationRef>
    </grounding:wsdlOperation>
    <grounding:wsdlInputMessage rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://127.0.0.1/wsdl/WeatherForecast/GetWeatherForecastInput
        </grounding:wsdlInputMessage>
    <grounding:wsdlOutputMessage rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://127.0.0.1/wsdl/WeatherForecast/GetWeatherForecastOutput
        </grounding:wsdlOutputMessage>
    <grounding:wsdlInput>
        <grounding:WsdlInputMessageMap>
        <grounding:owlsParameter rdf:resource="#_DATE"/>
        <grounding:wsdlMessagePart rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://127.0.0.1/wsdl/WeatherForecast/_DATE</grounding:wsdlMessagePart>
        <grounding:xsltTransformationString>None (XSL)</grounding:xsltTransformationString>
    </grounding:WsdlInputMessageMap>
    </grounding:wsdlInput>
        <grounding:wsdlInput>
     <grounding:WsdlInputMessageMap>
        <grounding:owlsParameter rdf:resource="#_CITY"/>
        <grounding:wsdlMessagePart rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://127.0.0.1/wsdl/WeatherForecast/_CITY</grounding:wsdlMessagePart>
        <grounding:xsltTransformationString>None (XSL)</grounding:xsltTransformationString>
     </grounding:WsdlInputMessageMap>
    </grounding:wsdlInput>
    <grounding:wsdlOutput>
     <grounding:WsdlOutputMessageMap>
        <grounding:owlsParameter rdf:resource="#_WEATHER"/>
        <grounding:wsdlMessagePart rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        http://127.0.0.1/wsdl/WeatherForecast/_WEATHER</grounding:wsdlMessagePart>
        <grounding:xsltTransformationString>None (XSL)</grounding:xsltTransformationString>
```

```
112     </grounding:wsdlOutput>
113 </grounding:WsdlAtomicProcessGrounding>
114
115 </rdf:RDF>
```

LISTING 2.3: WeatherForecastRequest OWL-S Service

Discussion

Since a WSDL document lacks declaring semantic data model that helps other machines to understand the meaning of document contents, we drop WSDL and consider choices (OWL-S and SAWSDL) on the semantic web service approach. Overall both OWL-S and SAWSDL can annotate elements of a WSDL interface with entities from a semantic data model, which facilitate automated service discovery and service composition. However, we choose OWL-S as service description for our Web service composition framework. As semantic web services that the Discovery system (DS) uses for its searching model is OWL-S services. Addition, OWL-S standards define ontologies for describing the capabilities and choreographies of stateful web services (choreographies being the sequences of messages exchanged between a client and a service during an interaction [DS12]).

2.3 Service Composition

In general, service composition is the method to create new services or applications by composing existing services. The output of service composition process is a composite service which delivers desired functions. Since users' requirements from the E-tourism scenarios are complex, an execution of sole service is unlikely to fulfill the users' needs. Thus service composition plays a part in the procedure of assembling the existing services if relevant to the users' needs.

We consider the service composition problem as the constructing business process problem in B2B interactions. Since both of them have a common goal to produce a structured and measured set of activities designed for a particular customer or market [Dav93]. As composite service and workflow have a common goal to build up a set of activities and services respectively to produce a specified output for a particular requirement, we use the terms composite service and workflow interchangeably. We also concern in this context of service composition is orchestration. The term orchestration model (or simply orchestration) refers to the

part of the composition that specifies the order in which the different component services should be invoked [ACKM04].

2.3.1 Web service composition life cycle

In this subsection, we discuss how a composite service can be built. Generally, service composition process occurs in three subsequent phases: composition, selection and execution (see in Figure 2.4).

- *Composition phase.* This phase deals with synthesizing the composition schema. Given a complex requirement, the composition schema designer decomposes the requirement to build up the composition schema or workflow schema. The schema consists of component services and control and data flow specification. The control flow specification sets up the order in which the component services should be invoked. Also the condition and/or timing constraints defined may be interrupt or cancel their execution. While data flow specification captures the flow of data between component services [BDFR03]. The composition schema can be constructed either manually at design time or automatically at run time. Note that if the composite service is written up from specific composition language such as eFlow [CS01], UML-WSC [TDE02] and BPEL [KMCW05], this composite service can later be executed automatically by a tied execution engine. Otherwise a transformation module is needed to convert composite schema from a graph or workflow model to executable composite services such as BPEL and BPMN language.

- *Selection phase.* In this phase, component services wiring in a composition schema are bound specific web services. The specific services can be discovered from service registry after the composite schema is formed. The selection engine finds and matches the advertised service specifications and the component service's functions. The result of this phase is an executable composite service. This service selection can be done either statically or dynamically. If the specific services are known in advance, alliances are statically defined. In other words, there is no service selection. The static approach, generally, works well as long as the web service environment such as business partners, service functionality and composite requirement do not

or rarely change. In contrary, if the selection of service components happen at run time, it will be classified as a dynamic services composition. In other words, the dynamic approach allows the execution system to support automated discovery, selection and binding of service components.

- *Execution phase.* In general, service execution governs the order in which services are invoked, and the conditions under which a certain service may or may not be invoked. In this phase, the executable composite service is deployed to create its instance. Next the composite service instance then allows an invocation by end user then is executed by process execution engine. The execution engine performs monitoring tasks. The tasks includes logging, execution tracking, performance measuring and exception handling. [SQV+14].

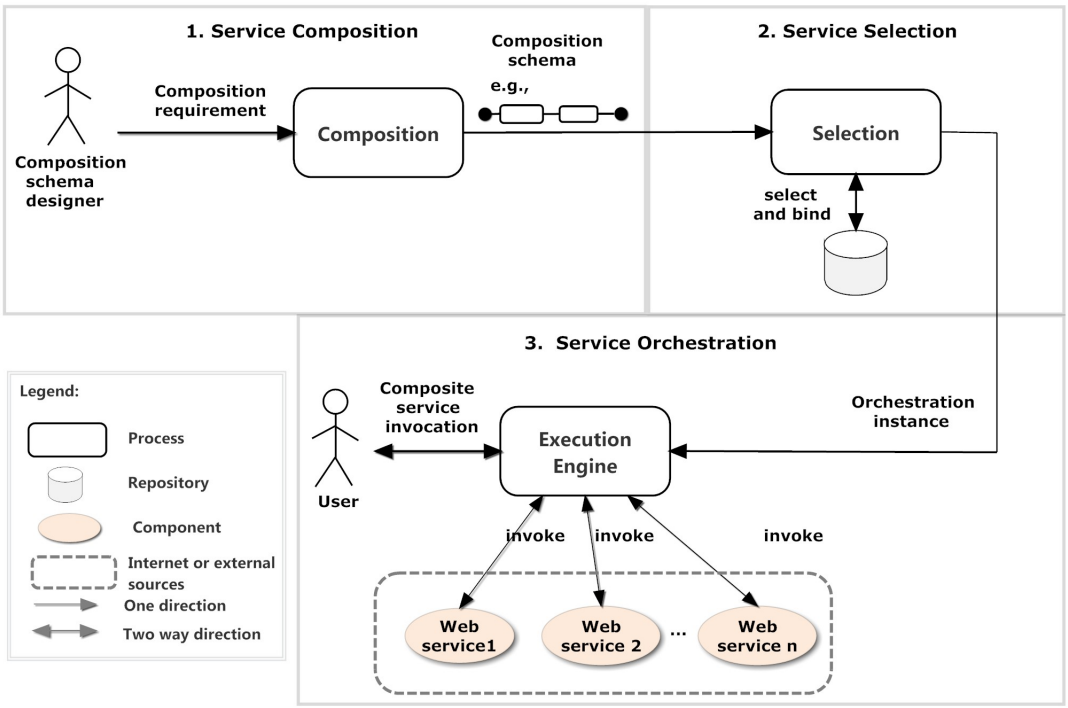


FIGURE 2.4: Web service Composition Life Cycle

2.3.2 Manual Approaches

In this section, we discuss a manual web service composition which can be occurred in Composition phase (see in Figure 2.4). Two different scenarios can be identified respectively into primitive level and abstract level.

In primitive level, the programmer uses business process languages, such as BPEL or OWL-S, to specify the composition schema. Consider the Listing 2.4 fragment BPEL code (line 1-103) of FindFinestRestaurant process. The process receives FindFinestRestaurantRequest given a coordinates from a client (line 52). Two web services (FindCity and FindRestaurant) are called in the process to deliver FindFinestRestaurantResponse to the client (line 59-100). With these business process languages, the programmer needs to specify both control flow and data flow of component services. As the result, the composition schema result is an executable composite process specification.

```
1  <?xml version="1.0" encoding="utf-8"?>
2
3  <!-- BPEL process -->
4
5  <process name="FindFinestRestaurant"
6           targetNamespace="http://packtpub.com/bpel/findFinestRestaurant/"
7           xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
8           xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
9           xmlns:ffrp="http://packtpub.com/bpel/findFinestRestaurant/"
10          xmlns:fcc="http://packtpub.com/service/findCity/"
11          xmlns:fr="http://packtpub.com/service/findRestaurant/" >
12
13    <partnerLinks>
14       <partnerLink name="client"
15                    partnerLinkType="ffrp:findFinestRestaurant"
16                    myRole="findFinestRestaurantService"
17                    partnerRole="findFinestRestaurantServiceCustomer"/>
18
19       <partnerLink name="FindCity"
20                    partnerLinkType="fcc:findCity"
21                    partnerRole="findCityService"/>
22
23       <partnerLink name="FindRestaurant"
24                    partnerLinkType="fr:findRestaurant"
25                    partnerRole="findRestaurantService"/>
26    </partnerLinks>
27
28    <variables>
29       <!-- input for this process -->
30       <variable name="FindFinestRestaurantRequest"
31                 messageType="ffrp:FindFinestRestaurantRequestMessage"/>
32       <!-- input for the Coordinates to City web service -->
33       <variable name="FindCityRequest"
34                 messageType="fc:FindCityRequestMessage"/>
35       <!-- output from the Coordinates to City web service -->
36       <variable name="FindCityResponse"
37                 messageType="fc:FindCityResponseMessage"/>
38       <!-- input for Restaurant web services -->
39       <variable name="RestaurantRequest"
40                 messageType="fr:RestaurantRequestMessage"/>
41       <!-- output from Restaurant -->
```

```
42        <variable name="RestaurantResponse"
43                  messageType="fr:RestaurantResponseMessage"/>
44             <!-- output from BPEL process -->
45        <variable name="FindFinestRestaurantResponse"
46                  messageType="fr:RestaurantResponseMessage"/>
47     </variables>
48
49     <sequence>
50
51        <!-- Receive the initial request for Find finest restaurant request from client -->
52        <receive partnerLink="client"
53                 portType="ffrp:findFinestRestaurant"
54                 operation="findFinestRestaurant"
55                 variable="FindFinestRestaurantRequest"
56                 createInstance="yes" />
57
58        <!-- Prepare the input for the Find City Status Web Service -->
59        <assign>
60          <copy>
61            <from variable="FindFinestRestaurantRequest" part="coordinates"/>
62            <to variable="FindCityRequest" part="coordinates"/>
63          </copy>
64        </assign>
65
66        <!-- Synchronously invoke the Find City Status Web Service -->
67        <invoke partnerLink="FindCity"
68                portType="fc:findCity"
69                operation="findCityFromCoordinates"
70                inputVariable="FindCityRequest"
71                outputVariable="FindCityResponse" />
72
73        <!-- Prepare the input for Finest Restaurant Web Service->
74        <assign>
75          <copy>
76            <from variable="FindCityResponse" part="city"/>
77            <to variable="RestaurantRequest" part="city"/>
78          </copy>
79        </assign>
80
81        <!-- Synchronously invoke the Finest Restaurant Status Web Service  -->
82            <invoke partnerLink="findRestaurant"
83                portType="fr:findRestaurant"
84                operation="findFinestRestaurant"
85                inputVariable="RestaurantRequest"
86                            outputVariable="RestaurantResponse" />
87
88            <!-- construct the FindFinestRestaurantResponse -->
89          <assign>
90              <copy>
91                <from variable="RestaurantResponse" />
92                <to variable="FindFinestRestaurantResponse" />
93              </copy>
94          </assign>
95
96        <!-- Make a callback to the client -->
```

```
97        <invoke partnerLink="client"
98                portType="ffrp:ClientCallbackPT"
99                operation="ClientCallback"
100               inputVariable="FindFinestRestaurantResponse" />
101     </sequence>
102
103 </process>
```

LISTING 2.4: FindFinestRestaurantProcessBPEL

Conversely, the composition schema is synthesized into an abstract workflow at abstract level. The abstract workflow determines control and data flow of component services in the abstract, without referring to any real services. For instance, the software designer uses UML activity diagram to build up an abstract service composite model. UML activity diagrams[19] are the most widely used process modeling paradigm, both in conventional middle-ware (workflow) and in web services. The reason for their success is that, orchestrations are defined by specifying which operations should be invoked, from the beginning of the execution to its end. This seems to be the most natural way in which people think of a process, and it is analogous to how developers code their application [ACKM04].

However, this manual approach is a time-consuming and not suitable for the Service composition and execution module in the E-tourism project (see section 1.1). The module aims to customize an application on-the-fly from existing services. Therefore we select the automated approach, which is described in the next subsection.

2.3.3 Automatic approaches

Generally, an automated approach generates a composite service by aggregating component services, without human intervention. To deal with the automatic approach, some approaches present web service automated composition with AI-planning techniques.

The concept of AI-planning is that planning can be interpreted as a kind of problem solving, where an agent uses its believes about available actions and their consequence, in order to identify a solution over an abstract set of possible plans [RN95]. Another definition of AI-planning based on state transition systems is

[19] http://www.omg.org/technology/documents/formal/uml.htm

that from deduction theory that the initial conditions together with the domain axioms (which define the semantics of the operators) and some sequence of actions imply the goal situation [Pee05].

We select AI-planning as planning technique to deal with the automated service composition problem. Figure 2.5 illustrates an overall process of automated service composition based on AI-planning techniques. Note that, inputs, processes and outputs in the diagram are represented in two different domains: problem domain written without parenthesis and planning domain written in parenthesis. Given user query, profile and context and a list of component services in problem domain are translated into initial and goal states and a set of actions respectively, an agent can synthesize an abstract plan (or composite service specification) using reference reasoning methods. As the result, the composition of web services is processed at run-time.

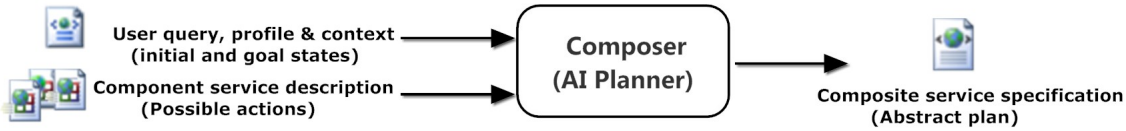


FIGURE 2.5: An overall process of automated service composition

Understanding components in AI-Planning domain is important. Since they affect a resulting plan. The classical view of a plan is a sequence of action instances, which its execution leads to a state that satisfies the user query. However, the classical plan might not be sufficient to capture the solutions to complex planning problem. If the agent does not complete information for constructing a plan, a conditional plan is required to deal with the possible solutions could arise. Besides the sequence and conditional plans, there exist several other extensions such as plans with parallel, branch and loop actions [Pee05]. An AI-planning approach generally defines a planning problem as the following components:

1. **A description of possible actions.** The possible actions (or planning domain) can represent both physical and abstract actions in the world. The possible actions (or planning domain) can represent both physical and abstract actions in the world. Let consider a trip planner robot in our E-tourism system whose task is to plan actions from a given constraints to achieve its goals. At the robot's physical environment, such actions can be defined: initial(X) for robot's initial state, goals(Ys) for a list of expected goals. To

achieve the mission, the trip planner robot must perform such abstract actions such as add_start, add_and and add_end. To specify these actions, a domain theory in some formal language is specified. A domain theory is a formal account of the semantics of the operations that are available or relevant to the agent [Pee05]. For example, the preconditions of actions and their effects to the world are defined. Classical logics such as situation calculus [Lin08], event calculus [KS89], model logics [CGLN01] are dominated to build up domain theories. Besides, the AI-planning community extensively formalizes languages such as Stanford Research Institute Problem Solver (STRIPS [FN72]) , Action description language (ADL [Ped94]) and Planning Domain Definition Language (PDDL [MGH+98]) to express planning domains. Among them, PDDL is the most recent and famous language in AI domain. The more expressive is the planning language, the richer and flexible is the resulting plan. For example, extended PDDL can express non-determinism and iterative conditional plans [CRB04].

2. **A description of the initial state of the world.** Besides the conceptual models of actions, a planning agent must take the initial world state into account. As it must provide a plan that, when executed in the initial world, will lead to the specific goal. From *booking a table at the finest restaurant* in Alice's partial query and her profile and context (see 1.2), initial(City), initial(GuestName), initail(NumberOfGuest) and initial(GuestTelephone) are defined as the initial world states. Generally, the initial world is just another world state defined by the domain theory [Pee05]. In the domain world, the definition of the initial world state provides a complete description. Thus, the resulting plan is a sequence of actions. While in the real world, we are confronted with the incomplete information. For example, in Alice's scenario, the agent may not know which restaurant offers the finest services in the city, however it needs this information to achieve its goal of booking a table. The solutions maybe are the planning agent deliver a conditional plan or a plan with branch action.

3. **A description of the desired goal.** In classic AI planning, goals are expressed as properties that need to be held in a desired world state. The planner needs to identify a plan which, when executed in the initial world state, will result in a world state that satisfied the goals. Consider again Alice's partial query of *booking a table at the finest restaurant*, its expected

goal in planning domain is goal(tableReservation). This goal indicates a condition that the goal of table reservation must exist after the plan execution. However, the real world situation, which is a complex problem, might express multi goals, solutions for nondeterminism problems and/or offers over preferences of users. This is important to provide the planning agent with domain or task dependent control knowledge in order to achieve good performance in real world domains [Pee05].

4. **Representing Plans.** Classical AI planning approach assumes that the initial world state is completely described and all actions are deterministic. Consequently, the planning agent synthesizes the sequence of actions implying the goal. However, this assumption is unrealistic for certain reasons. First, there is a situation that the information received from user entry or user context does not specify all knowledge relevant to the planning task. For example, in Alice's query that *she wants to book a room for 3 nights*, the agent might not know which hotel has available rooms for a given period, but it needs this information to achieve its goal of booking a room. Second, the execution of some operations does not meet the expected or desired result. We call these actions as non-determinism actions. Third, users sometime may want to specify their preferences or constraints on the solutions. For instance, back to Alice's partial query that *she wants to go to Louvre museum if the forecast weather is sunny.*

   These situations confronted with incomplete information or user preferences and constraints or non-determinism actions could lead the agent to construct a *conditional* plan, which generate for the possible branch plans [Pee05]. Thus, modern plans need complex control structures such as loops, non-determinism and condition [SK03]. Furthermore, we are also interested in planning that includes concurrency in service access for more efficiently execution time. These are the criteria of our proposed system. We review planning techniques regarding planning with control knowledge in section 2.3.4.

To benefit from existing AI-planning systems, an encoding of composite requirement and a set of operations in problem domain into possible actions, initial condition and goal situation in planning domain is required.

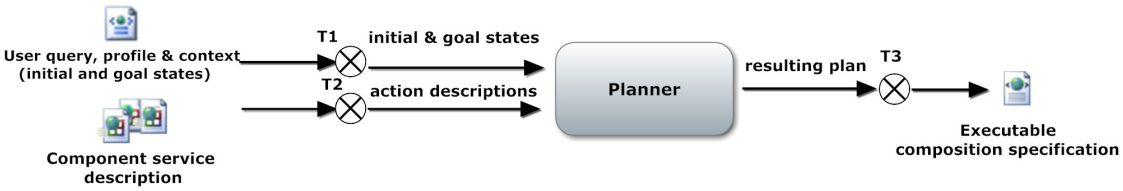


FIGURE 2.6: Domains transformation in automated service composition process

Figure 2.6 shows three transformations notated as ⊗ that can occur in automated service composition process. The T1, T2 transform problem domain (user query and constraint and component services specifications) into planning domain (initial and goal states and action descriptions). For example, Alice's partial query of *booking a table at the finest restaurant* and her profile and context are converted to planning domain as initial(city), initial(guestName), initail(numberOfGuest) and initial(Telephone) goal(tableReservation). The T3 occurs when we want to convert an generated abstract plan which cannot be directly executed by any process execution engine into executable composition specification. For example, T3 transforms a workflow represented in activity diagram to BPEL code. Later, BPEL process can be executed by BPEL engine.

The latest transformation brings us the advantage of flexibility of process execution. Since an abstract plan in the composite activity can be transformed into more than one executable description [LOKX13].

2.3.4 Existing Approaches

Many service composition techniques have been proposed in the literature (see for example [MM04, RS04, DS05, AEE06, LOKX13, SQV+14]). To analyze them, we structure this section into three topics. The first one relates to design approaches for end-to-end composition framework. The second and third topics discuss work related to automated logical composition and business process generation respectively.

2.3.4.1 End-to-end composition framework

We propose an end-to-end composition of web services. We formulate the web service composition problem and describe the integrated system for composing

web services from specification to deployment. Not many previous works implemented the end-to-end composition of web services. They concerned only logical or physical composition.

However, there is a METEOR-S system, which provides end-to-end composition framework [POSV04]. The framework uses the notation of a service template derived from ontological concepts. A service template refers to the description of a single web service which consist of a set of operations with their inputs and outputs [MMV+05]. The output of logical composition is an abstract BPEL which does not refer to any invocable service instances but semantic which does refer to semantic type. This compares to us, we generate abstract Business Process Modeling Notation (BPMN) which can support sequential and branches composition, while abstract BPEL generates only sequential composition.

Another end-to-end service composition system is Synthy [ACD+05]. The system uses an ontological concept for differentiating between service type and service instance. Service types represent groupings of similar web services while service instances refer to the actual web service instances that can be invoked. The planner generates a plan which has sequence, choice and concurrency among actions. The abstract plan later is transformed into abstract BPEL. Next the service discovery component picks exactly one instance from the set of matching instances for each service type. The Instance selector process takes into account global optimization criteria (e.g., QoS). Two main differences with our work are that: (a) Synthy has its service selection process after its service composition process. Compare to us; the selection of relevant services is done (see from [CC14]) before the service composition. (b) Synthy does not validate a translated abstract BPEL whether it is well formed BPEL specification or not.

2.3.4.2 Proposals of logical composition

At automated composition level, several service composition frameworks toward AI have been reviewed [McD00, MS02, NAI+03, BCG+05, CSHG09, BPT10]. Due to web services are considered in the context of workflow [DPAM07] that considers web services in the context of workflows, the problem of automated composition of workflow tasks can be seen as an Artificial Intelligence planning problem.

This subsection discusses the existing approaches regarding planning with control knowledge that can be used to compose web service automatically. We then compare these approaches to justify our workflow construction strategy.

PDDL based. The approach of McDermott use an extended PDDL as formal language to formalize web services. He translates DAML-S (former version of OWL-S) specification into PDDL action specification before executing a planner. The planner is then given an initial situation, a set of action definitions, and a goal to be carried out. A solution is a sequence of actions that, when executed beginning in the initial situation, carry out a situation in which the goal is true [McD00]. In addition, the planner generates based on estimated-regression conditional plans. In comparison to our criteria, McDermott extends PDDL to support neutral and rich specification of planning problems. His approach deals with the nondeterminism and incomplete information. However, the resulting plan has only sequences and conditions among actions whereas our resulting plan supports sequences, conditions and concurrencies among actions.

SHOP2. Simple Hierarchical Ordered Planner 2 (SHOP2) is a well known domain-independent planning system based on Hierarchical Task Network (HTN) planning [NAI+03]. HTN is an AI planning methodology that creates plans by task decomposition.

One of classical works on automated web service composition using SHOP2 is the works in [SPW+04]. The authors show how a set of OWL-S service descriptions can be translated to a planning domain description that can be used by SHOP2. The SHOP2 planner executes a given service composition problem by decomposing recursively the task into sub-tasks. The execution stops when the composite service contains only primitive operations. The resulting composite service is a sequence of service operation calls that can be subsequently enacted. The approach is also capable of executing web services for information-providing during the planning process [SPW+04].

However, the generated plan from SHOP2 has only sequence among actions. This hierarchical planning is suitable for Plan-management problems in which the plans tend to consist of abstract structures of actions [McD00].

Golog. The Golog is a logic programming language built on top of the situation calculus [LRL+97]. The approach that uses Golog to solve automated web

service composition is proposed in [MS02]. The authors set up service composition problem as a set of atomic actions, which are derived from an OWL-S ontology of services and client request, which is a skeleton of Golog procedure (constructed with sequence, choice and so on) expressing also client constraints and preferences. The approach differentiates between knowledge self-sufficient and physically self-sufficient. As the result, they propose ConGolog interpreter that combines on-line execution of information providing with off-line simulation of world altering web services. A number of different approaches have been discussed [GL99, Lak99, Rei01]. The planner with on-line interpreter can determine a sequence of web services for subsequent execution [GL99, Rei01] while the conditional plans can be generated by the use of an off-line interpreter [Lak99].

Solving web service composition with Golog is sound and complete as it presents good results for the problems of planning with non-deterministic actions, partial observations of the world, concurrency actions and non-linear plans. However, systems implemented with Golog are semi-automated approaches; since web service composition problem is predefined using requirement templates.

MBP. Model checking is a formal verification technique used to determine whether or not a property holds in a finite state model [DPAM07]. Therefore, Planning as Model checking paradigm is that planning problems should be solved model-theoretically [GT00].

Various publications adopt a concept of *Planning as Model Checking* (see examples in [MPT08] and [BPT10]). Given a description of component services and client service (e.g. in Abstract BPEL format) and composition requirements (e.g. the global goals including control flow and data flow), the planner synthesizes automatically the composite service that implements the internal process. Since this approach applies symbolic model checking, the composite service is automatically monitored to detect whether the component services behave consistently with the specified protocols. The approach shows a good practical result for the problem of planning with stateful service, non-deterministic actions, partial observations of environment, complex goals and domain [DPAM07].

TABLE 2.1: Characteristics of existing AI planning system

Systems	**Planning domain**	**Goal spec.**
PDDL based	PDDL atomic actions	propositional logic
SHOP2	SHOP2 atomic and complex actions	task name
Golog based	atomic actions in Situation Calculus	Golog procedure
MBP	abstract process in STS	global requirement (control and data flow)
FLUX based	atomic actions in fluent calculus	propositional logic

Our approach develops the idea of model checking to monitor a semantic model of composite service and also check a well-formed and well-defined of business process which is later executed.

FLUX. The FLUX (stands for: Fluent Executor) is a logic programming language for the design of intelligent agents that recognize about their actions using the fluent calculus [Thi98]. FLUX is used to solve an automation service composition problem [CSHG09]. The approach models a set of actions for web service composition by translating OWL-S service description to fluent calculus formalization. Then given initial and goal states of composite requirements, the FLUX planner generates a workflow of relevant actions. However, the workflow has only a sequence control among actions.

Recently, another work proposes a composition and verification framework for semantic web services specified using Web Service Specification Language (WSSL), a novel specification language for services, based on the fluent calculus [BP14]. The framework is implemented using FLUX-based planning, supporting compositions with control constructs such as conditionals and loop. However, this is similar to Golog approach, which the possible generic procedure for particular problems are needed to predefined.

According to the information above, we summarize characteristics of the existing AI Planning systems and a comparison among these automated planner systems in Table 2.1 and 2.2 respectively.

Discussion

So far, we have presented the comparison among existing planner systems in table 2.2. The following describes five essential criteria of plan characteristics.

TABLE 2.2: Comparison among planner systems

Plan characteristics	**PDDL**	**SHOP2**	**Golog**	**MBP**	**FLUX**
Non-determinism	Y	Y	Y	Y	Y
Extended goals	N	Y	Y	Y	Y
Generation of non-liner plans	Y (partially)	Y (partially)	Y	Y	Y
Concurrency	N	N	Y	Y	Y
Automation level	A	A	SA	A	A

- **Non-determinism.** This refers to a situation when an agent is given incomplete information to accomplish its goal. For example, the agent is assigned to booking a table in a restaurant to Alice. However, Alice does not know which restaurant yet. Therefore, the agent needs to solve it to find the restaurant.

- **Extended goals.** This refers to a situation when an expected plan contains many goals. Since in the reality, users' behavior and need are complex, the planner should be able to handle this situation. For example, from Alice's query that she wants to buy ticket of Louvre museum if the forecast weather is sunny or go for Disney Euro land instead. Two goals (ticket of Louvre museum and ticket of Disney Euro) are implicit in this example.

- **Generation of non-liner plans.** Due to presenting of non-determinism and extended goals, a planning agent constructs a conditional plan, which accounts for the possible branch plans. For example, the resulting plan for the above mentioned Alice's query is the conditional plan that has two alternative paths; buyDisneyEuroTicket operation is called if value of weather is sunny; buyLourveMuseumTicket operation is called if value of weather is not sunny.

- **Concurrency.** Generally, an action can be fired when its input(s) is existed. So if all inputs of relevant actions are hold in the current environment, the actions can be executed at the concurrency time. This make overall plan execution more efficiently. For example, renting a car: information of renting a car $\mapsto$ renting car receipt operation and booking a hotel room: information of booking a room $\mapsto$ booking room confirmation operation can be executed

at the same time as none of linked parameters is found between these operations.

- **Automatic level.** We interest in automated level of plan generation by the planning agent. For full automated level, the agents are able to design control flow automatically. While semi-automated level, the abstract workflow is derived from composition requirements template.

From the comparison shown in Table 2.2 and 3.3, all planner systems support nondeterminism actions. However, system based on PDDL and SHOP2 only allow conditional plans and do not support concurrency among actions. Only Golog allows semi-automated composition. We have noticed that the more expressivity of planning domain, the more complex of composition solution you could get. Thus, MBP is the best alternative with respect to the generation of non-liner plans as stateful services. However, MPB is tight with abstract BPELs for service components and client, this limits a number of BPEL service partners.

We choose fluent calculus to represent a description of component services. As fluent calculus is formalism language for reasoning and planning actions in dynamic environment. The more important is that fluent calculus works compatibly with FLUX planner. The planner provides logical constructs for assembling primitive actions into complex actions. FLUX complex action constructs include concurrence, conditional, sequential and nondeterministic actions [WNI+09]. We will give more detail on fluent calculus and FLUX later in the next chapter.

2.3.4.3 Business process generation and execution

As our ultimate goal is to execute the generated composite service and send the result of service execution to user, the resulting plan, which is a template for the composite service, needs to be in a format that can be executable. Thus, business process generator transform the abstract plan (or abstract workflow) into a standard business process model.

Many efforts, such as WS-BPEL[20], WS-CDL[21], BPMN[22] and OWL-S[23], have been underway to define standards for composing web services. In addition,

[20]http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf
[21]http://www.w3.org/TR/ws-cdl-10/
[22]http://www.bpmn.org/
[23]http://www.w3.org/Submission/OWL-S/

many software enterprises offer business process programs based on business process language standard. The tools basically consist of business process modeler and executor. To benefit business process standards, many existing works [SPW+04, ACD+05, BCG+05, BPT10] convert their abstract plan into abstract business process, which later drives the process of matching each process task to a corresponding service instance.

SHOP2 system, discussed in the logical composition, offers the system output as business process [SPW+04]. The output of SHOP2 system is a sequence of world-altering web service calls in SHOP2 domain. The OWL-S plan converter converts the plan produced by SHOP2 to OWL-S format which can be directly executed by an OWL-S executor. The OWL-S web service executor which communicate with SOAP-based web service description by OWL-S grounding to WSDL description of those services.

While many works adopt BPEL standard in their automated web service composition (see in [ACD+05, BCG+05, BPT10]). [ACD+05] presents Synthy web service composition tool which use Planner4J[24] to generate contingent plans. The Planner4J generates output plan in workflow (BPEL4WS) format. Works in [BCG+05, BPT10] use state transition system (STS) in their logical composition. With the expressive of STS, the resulting plan supports full control structures such as loops, conditionals and parallel. Each work proposes algorithm for the translation of such synthesized STSs to BPEL.

2.4 Summary

So far, we have been given the background and related work regarding to automated service composition system. We start from interactions in B2B which occur in three layers: Communication, Content and Business process layers. To fulfill the contents for communication and content layers, we mention Service-Oriented Architecture (SOA) and web services. Web services is the most popular and complete implementation of SOA. Two semantic web services description language (SAWSDL and OWL-S) are introduced since they are needed for automated service composition system. However, we select OWL-S for component services as OWL-S supports conceptual markup of web services and process model which

[24] http://researcher.watson.ibm.com/researcher/view_person_subpage.php?id=914

describes the capablities and choreographies of stateful web services. Then we explain Web service composition life cycle for the business process layer. We an emphasize automated service composition and then introduce AI planning as a planning technique to deal with the automated service composition problem. For the related work, we have divided automated service composition into 3 levels: framework, logical composition and business process generation levels.

Chapter 3

Abstract Service Composition with Fluent Calculus

Contents

Abstract. An automated workflow of Abstract service composition system, one of our contributions, is introduced in this Chapter. Since we see service composition problem as planning problem, we separate the system into two processes: Transformer process and Planner process. The Transformer changes the user's context and query into planning problem and transforms

> a list of service operations into a set of possible actions. The Planner shall try to reason the updated state of the given actions to obtain a workflow of actions, which satisfies the planning problem.

This chapter presents the Abstract Service Composition process which is the process present in Logical layer of the Service Composition and Execution system (see in Figure 3.1).

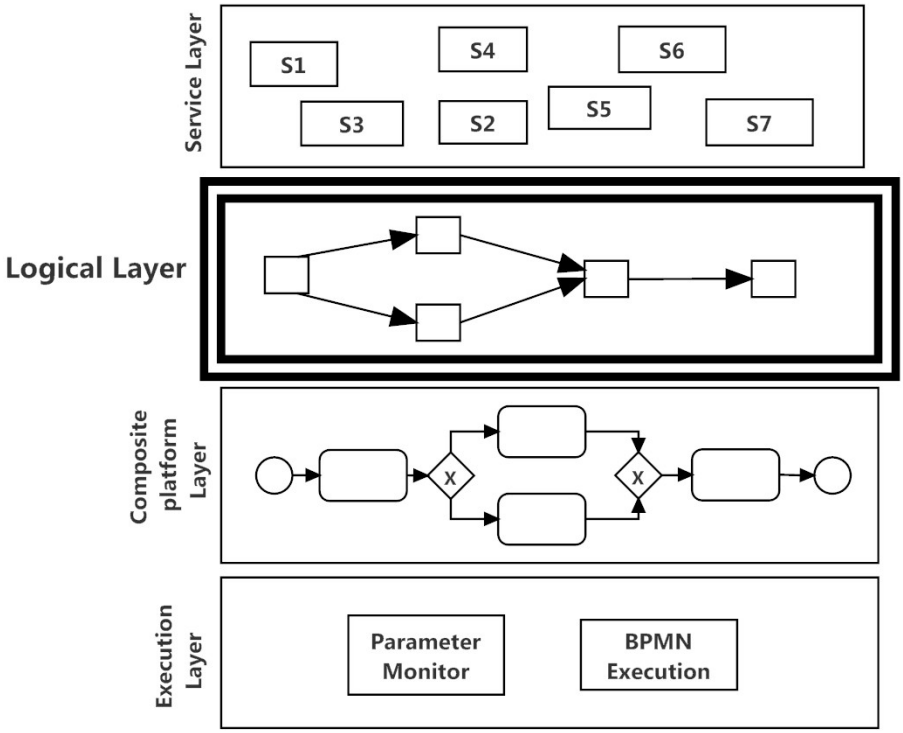


FIGURE 3.1: Proposed multi-layers of the Service Composition and Execution framework highlighted at the Logical layer

The aim of the Abstract Service Composition process is to synthesize an abstract plan, which is constituted from operations offered by services. It is an automated approach where there is no human intervention during the service composition process. We modeled user's initial, user's query and service operations into fluent calculus and use FLUX Planner, which is constraint logic programming, to draw the expected output plan. Nevertheless, the FLUX planner reasons the updated state of the given operations to obtain the workflow of operations (so called abstract plan or abstract service composition) which satisfies the user's goals and conditions.

This chapter is organized as follows: requirements and architecture of the Abstract Service Composition system discussed in Section 3.1. Section 3.2 presents a

motivating example used throughout the chapter. Then we propose Transformer process and FLUX Planner process in Section 3.3 and Section 3.4 respectively. Lastly, we compare our service composition approach to the existing works in Section 3.5.

3.1 Requirements and Architecture

To develop our Abstract Service Composition system, we list the overall system requirements as following:

1. System interacting with the Discovery system to receive system inputs: user's query (expressed in terms of goals and initials), her profile, context and a list of relevant service description operations.
2. System transforming the user's query, his or her profile, context and service operations (in requirement no.1) into initial and goal states and possible operations in planning domain respectively.
3. System reasoning these service operations (in requirement no.2) to generate an abstract plan.
4. The resulting abstract plan satisfies all expected goals and initial constraints in requirement no.3.
5. The resulting abstract plan may consist one or more link controls such as sequence, condition and parallel among operations.

From the above stated requirements, we separate the Abstract Service Composition system into two components: Transformer and FLUX Planner as depicted in Figure 3.2. The Transformer component is responsible for pre-processing inputs from problem domain into the user's planning domain. While FLUX Planner component is a service operation composer, which performs assembling service operations to get the desired abstract plan.

The user's query along with her profile, context and service operations in problem domain shall be considered as initial, goal states and possible actions in FLUX query respectively. The FLUX query shall be solved by the FLUX planner. We use AI-planning techniques to implement our FLUX planner to perform automated

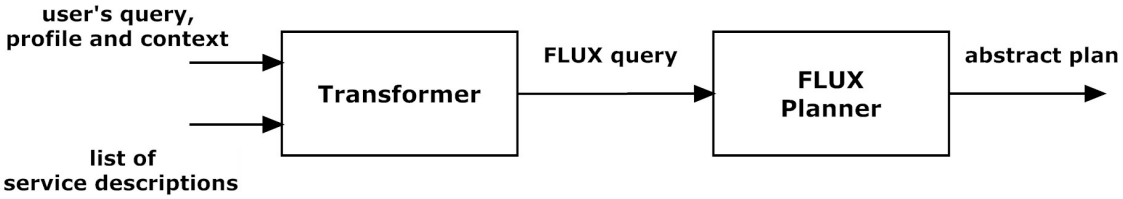


FIGURE 3.2: Abstract Service Composition process

abstract service composition. There are many previously approaches adopting AI-Planning methods to solve automated service composition. We have reviewed and compared those existing in Section 2.3.4.2, Chapter 2.

The reason we chose FLUX over other AI systems because FLUX is implemented based on fluent calculus[1]. Modeling user's constraints and relevant operations with fluent calculus, the FLUX Planner reasons on a variety of operations such nondeterministic, conditional and concurrency to obtain a resulted abstract plan. The rest of this section is dedicated to fluent calculus and FLUX as a background for the Transformer and FLUX Planner components in Section 3.3 and 3.4 respectively.

3.1.1 Fluent Calculus

Fluent Calculus is a language for reasoning and planning actions in dynamic environment [Thi98]. It is a variant of the situation calculus. The main field using the fluent calculus is in autonomous robotics which models state of the world before and after executed robot actions. The objective is to represent non-effect actions and infer them under the frame problem. The frame problem is the problem of finding adequate collections of axioms for a viable description of a robot environment [Hay71].

One of the research projects realizing fluent calculus is ALAN (Axiomatization Language for Agents). The authors designed an axiomatization language for autonomous agents and mobile robots [Thi02]. This aims to enable the robots to determine the response actions it must decide with respect to the knowledge given in the effects of these actions [Thi02]. A question may be raised: how such actions are designed and described?

Generally, four sorts of entities based on fluent calculus construct the dynamic actions in agent's specific environment. The fundamental entity is a *fluent f*, which

[1]A formal mathematic expression for dynamical domain in first order logic.

is a single atomic term f of the physical world, which may change on time [BP14]. For example, the robot's position at location (x,y) is defined in fluent calculus as $f = At(x,y)$. The second entity is a *state*, which is a collection of fluents. In other words, states are built up from fluents and their conjunction, using the function $\circ$*: state, state* $\mapsto$ *state* [Thi05]. For instance, the initial state of Alice's query that *she wants to book a table at restaurant tonight* can then be formalized by this term:

$$Restaurant(r) \circ Guest(g) \circ Date(d) \circ TableAvailable(t) \tag{3.1}$$

The third is an *action*, which is a high-level action referring to things that can change the world state. Finally, a *situation* is a sequence of actions.

In agent programs, conditions which refer to the state of the outside world are based on the notion of fluents to *hold* in states [Thi05]. For example, it is impossible for the planner to book a table for Alice if fluent *Restaurant(r)* does not hold in the current state. The method to assign any *fluent f* into a *state z* is presented in formula (3.2). It indicates that z can be identified with f that hold in it and z can be decomposed into f and some state z' [Thi05].

$$Holds(f,z) \equiv (\exists z' \text{ is a state}, \circ(f,z') = z) \tag{3.2}$$

Therefore, $(\exists r)$ *Holds(Restaurant(r),z)* where z is the state in (3.1) is equivalent to *Restaurant(r)* plus some arbitrary other sub-state: $Guest(g) \circ Date(d) \circ TableAvailable(t)$.

There are two predicate formulas in the fluent calculus offering functions to hold a state z in an action before and after state execution. The first one is an *action precondition axiom* for formally specifying the circumstances under which an action is possible in a state. With action functions $\mathbb{A}$ and let $A \in \mathbb{A}$, a *action precondition axiom* for A is a formula:

$$Poss(A(\vec{x}),z) \equiv \Pi(z) \tag{3.3}$$

where Π(z) is a state formula with free variables among $\vec{x}$, z. The formula explains that the action A is possible at state z, if and only if a state formula Π(z) is true. For instance, precondition states of booking table at restaurant action can be

formalized by this axiom:

$$\begin{aligned}&Poss(BookingTable(r,g,d),z) \equiv\\&Holds(Restaurant(r),z) \wedge (\exists t)(Holds(TableAvailable(t),z)) \wedge\\&Holds(Guest(g),z) \wedge Holds(Date(d),z).\end{aligned} \tag{3.4}$$

With this specification, in the initial state (3.1) this action is possible. Another predicate is a *state update axiom* for formally specifying a resulting state that an action caused. With action functions $\mathbb{A}$ and let $A \in \mathbb{A}$, a *state update axiom* for A is a formula:

$$Poss(A(\overrightarrow{x}),s) \supset (\exists(\overrightarrow{y}))(\Delta(s) \wedge State(Do(A(\overrightarrow{x}),s)) = State(s) + \theta^{+} - \theta^{-}) \tag{3.5}$$

where Δ(s) is a situation formula with free variables among $\overrightarrow{x},\overrightarrow{y}$,s; θ^{+}, θ^{-} are finite states with variables among $\overrightarrow{x},\overrightarrow{y}$. The formula shows that if conditions of action A is possible at situation s, executing it results in a successor state derived for state(s): positive effect θ^{+} and negative effects θ^{-}, under additional conditions Δ(s) [Thi98].

For Instance, state update of booking table at restaurant action can be formalized by this axiom:

$$\begin{aligned}&Poss(BookingTable(r,g,d),s) \supset\\&(\exists t)(Holds(TableAvailable(t),z)) \wedge\\&State(Do(BookingTable(r,g,d),s)) =\\&State(s) - TableAvailable(t) + TableBooked(r,g,d,t)\end{aligned} \tag{3.6}$$

With this specification, $State(S_0)$ = (3.1), then $Poss(BookingTable(r,g,d),S_0)$ according to the precondition axiom in (3.4). Let $S_1 = Do(BookingTable(r,g,d),S_1)$, then the state update axiom for booking table action implies:

$$State(S_1) = State(S_0) - TableAvailable(t) + TableBooked(r,g,d,t)$$

A solution to this equation is given by:

$$State(S_1) = Restaurant(r) \circ Guest(g) \circ Date(d) \circ TableBooked(r,g,d,t)$$

3.1.2 FLUX

With state representation in dynamic environment using fluent calculus, the constraint logic programming called FLUX (Fluent Executor) is developed. The FLUX aims to design of agent programs that reason about their actions using the fluent calculus [Thi98].

These agents use the concept of a state of the world when controlling their own behavior. Regularly, agents update their state of the world to reflect the changes of their actions' effect and/or obtaining sensor information. The structure of FLUX consists of three modules: P_k, P_d and P_s. P_k is a set of constraint handling rules and constraint solvers for finite domain. P_d contains encodings of the domain axioms including action precondition axiom, update axioms, domain constraints and initial knowledge state. Lastly P_s specifies a high level of specific strategy on the behave of the agent it will perform. Due to the powerful constraint solver and the underlying FLUX kernel prover general reasoning facilities in P_k, the agent programmer can focus only on specifying the application domain in P_d and designing the high-level behavior in P_s [Thi98].

Overall, FLUX performs outstanding linear computational behavior due to its inference engine. Additional, applying the progression principle, FLUX scales up well to long-term control [Thi98].

3.2 Back to the motivating example

At four in the afternoon, Alice wants to book a table for 2 people at the finest restaurant at 8pm, in the city, and also find the direction. She uses the E-Tourism system (see Section 1.1) via her smart device and submits a query: *I want to book a table for 2 people at the finest restaurant at 8pm in the city, and I want to know the direction to the restaurant.* Alice's device sensors also capture her context information she submitted the query that coordinates = "48.8567° N, 2.3508° E" and date = "1/06/2014". The system has knowledge of Alice's profile that name = "Alice", citizenship="USA", travelPurpose="work", preferred activities="outdoor", gender="female". For privacy reason, Alice wants to hide her location so the User Privacy module dose not disclose her location information out from her context. These data from three sources (query, context and profile) are forwarded to the

User Management system to discover the parameters of the query and split the query into sub-queries. Later the sub-queries and discovered parameters are sent to the Discovery system. The Discovery system shall send to the Composition and orchestration system a list of candidate services, each of which corresponds to the spitted sub-queries, and the forwarded parameters. Then the Composition and orchestration system compose operations offered by the services to fulfill the query. The resulting service to be executed is the composition of the operations below:

FromCoordinatesToCity: Coordinates → City
FindFinestRestaurant: City → RestaurantID, RestaurantName, RestaurantAddress
BookRestaurant: RestaurantName, BookingTime, BookingDate, NumberOfGuest, GuestName, GuestTelephone → BookingRestaurantReservation
GetDirection: FromAddress, ToAddress → Direction

3.3 Transformer to fluent calculus

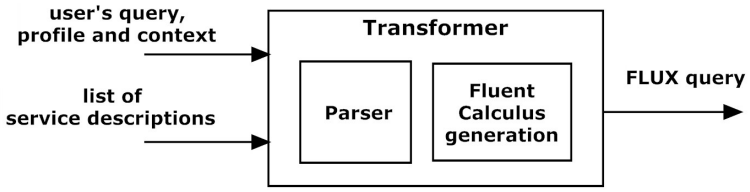


FIGURE 3.3: Transformer process

The Transformer is the process in the Abstract Service Composition system. It intents to pre-process data from user problem domain into initial knowledge state and constraints in planning domain. The data from user problem domain consists of parameters extracted from the user's query, the profile, the context and operations offered by service descriptions. In Figure 3.3, the transformer parses the parameters and the service operations and then transforms them into initial and goal states and operations in fluent calculus axioms respectively. Their converted results are combined for a FLUX query. The Flux query is then forwarded to FLUX planner. The following section presents the transformer process with the motivating example (see Section 3.2), which is subjected to user requirements and service operations mappings.

3.3.1 User requirements mapping

This subsection shows what are the user's requirements and the parameters of the user's goals and her profile and context, are transformed to initial and goal states. Consider user requirements mapping in Table 3.1 as an example:

User problem domain		**Planning domain**
Query	RestaurantName(goal) BookingReservation(goal) Direction(goal) NumberOfGuest(2) Time(8pm)	*goals*([RestaurantName, BookingReservation, Direction]) *initial*(NumberOfGuest) *initial*(Time)
Context	Date(01/06/2014)	*initial*(Date)
Profile	Name(Alice)	*initial*(Name)

TABLE 3.1: Example of a mapping between user problem domain and planning domain

The parameters classified into query, context and profile categories are shown in user problem domain column. We use the convention for each parameter as name of parameter followed by its value beside parenthesis. The value itself has two options: known value or goal value. The known values are extracted from user queries, disclosed context and profile information. For example, NumberOfGuest(2), Time(8pm), Date(01/06/2014). The goal values are the unknown values of things the users want to possess or achieve. For instance, RestaurantName(goal), BookingReservation(goal), Direction(goal).

To map parameters from problem domain to fluents in planning domain, we shall follow these two rules: 1) Parameters having the goal values are grouped into a list of goals fluent. For instance, goals([RestaurantName, BookingReservation, Direction]). 2) Individual parameter having known values is converted to initial fluent. For example, initial(NumberOfGuest).

3.3.2 Service operations mapping

Besides user requirements mapping, the transformer converts service operations into input fluent operations. Each operation has its naming conventions of *op_inputs* and *op_outputs* clauses. For each service operation, we have only one clause *op_inputs* and one clause *op_outputs*. Generally, *op_inputs* and *op_outputs*

refer a set of operation input and a set of operation outputs. Consider service operation mapping in Table 3.2 as an example:

Operations in Problem domain	FindFinestRestaurant input: City outputs: RestaurantName, RestaurantAddress
Operations in Planning domain	*op_inputs*(FindFinestRestaurant, [City]) *op_outputs*(FindFinestRestaurant, [RestaurantName,RestaurantAddress])

TABLE 3.2: Example of service operations mapping between problem domain and planning domain

It's a worth noting that we separate input and output operations into different naming conventions. It enables the Planner to synthesize the control links among operations in the abstract plan. More details about the Planner are given in the next section.

3.4 FLUX Planner

Planner is a process happening after the transformer process in the Abstract Service Composition system. It intents to synthesize a plan from abstract operations to fulfill the user's goals. According to the system requirements (see Section 3.1), outcome of this resulting plan should satisfy all expected goals and initial constraints and also support linked control constructs such as sequence, condition and parallel.

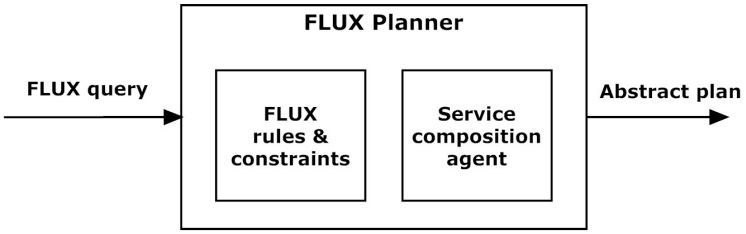


FIGURE 3.4: FLUX planner

To achieve this operation, we propose FLUX planner, which is a constraint programming based on fluent calculus (see Subsection 3.1.2 for more details) see its architecture depicted in Figure 3.4. The FLUX planner consists of three components: (1) FLUX library, (2) FLUX query and abstract plan and (3) service composition agent.

The FLUX library contains a set of constraint handling rules and a constraint solver for finite domain of our FLUX planner [Thi05]. The FLUX query contains encodings of the domain axioms including condition constraint, initial and goal knowledge state and a list of service operations, while the abstract plan contains encoding of the plan axioms including initial and goal nodes, flow of composable operations and data transformation. The service composition agent performs agent's actions including action precondition axiom and update axioms, under the behavior of planning composition.

3.4.1 FLUX query and Abstract plan

To achieve synthesizing the resulting plans defined in Section 3.1, we develop planning model for the service composition agent. The model is formulated using fluents for the FLUX query and the abstract plan structures. We use the special sorts `OPERATION, PARAMETER, VALUE` and `INDEX` along with the fluents. One instance of FLUX query necessarily consists of following basic fluents:

```
Initial(X): PARAMETER --> FLUENT
Goal(Xs): PARAMETER --> FLUENT
Op_inputs(X,Ys): OPERATION x SET OF PARAMETER --> FLUENT
Op_outputs(X,Ys): OPERATION x SET OF PARAMETER --> FLUENT
```

`Initial(X)` is a fluent for initial parameter `X` (i.e., *initial*(RestaurantType) and *initial*(Coordinates)). `Goal(Xs)` is a fluent for a set of goal paramaters `Xs` (i.e., *goals([RestaurantName, BookingReservation, Direction])*). `Op_inputs(X,Ys)` is a fluent for a set of input parameters `Ys` of a single operation `X` (i.e., *op_inputs(FromCoordinatesToCity, [Coordinates])*) and `Op_outputs(X,Ys)` is a fluent for a set of output parameters `Ys` of a single operation `X` (i.e., *op_outputs(FindFinestRestaurant, [RestaurantName, RestaurantAddress])*). Besides these basic fluents, we have fluents for representing user constraints in the FLUX query. For example:

```
Cond(C,if(guard(O1,RO1,V1),O3),else(guard(O2,RO2,V2),O4)):
INDEX x PARAMETER x OPERATOR x VALUE x PARAMETER x
PARAMETER x OPERATION x VALUE x PARAMETER --> FLUENT
```

`Cond(C,if(guard(O1,RO1,V1),O3), else(guard(O2,RO2,V2),O4))` is a fluent for condition guard request. The index `C` request contains two alternative paths if the first guard condition indicating parameter value O1 with relational operation RO1 equals to value V1 is *true* then parameter O3 exists; else if the second guard condition saying parameter value O2 with relational operation RO2 equals to V2 is *true* then parameter O4 exists. For instance, *cond(c1, if(guard(weather,==,sunny),ticketD),else(guard(weather,==,rainy),ticketM))*. It's worth noting that `DataTransform(X,X1)` fluent might be hold if there is transformation of data passing between parameter `X`, where `X` is a subset of operation outputs and parameter `X1`, where `X1` is a subset of another operation inputs.

We show a complete FLUX query specification as example in Listing 3.1 for initial and goal states and input operations, which is relative to Alice's scenario (see Section 3.2).

```
1  init(Z0) :-
2
3  Z0 =
4  [op_inputs(fromCoordinatesToCity,["Coordinates"]),
5  op_inputs(findFinestRestaurant,["City"]),
6  op_inputs(bookRestaurant,["RestaurantName","BookingDate","BookingTime","NumberOfGuest",
7  "GuestName","GuestTelephone"]),
8  op_inputs(getDirection,["Coordinates","Address"]),
9
10 op_outputs(fromCoordinatesToCity,["City"]),
11 op_outputs(findFinestRestaurant,["RestaurantName","RestaurantAddress"]),
12 op_outputs(bookRestaurant,["BookingConfirmation"]),
13 op_outputs(getDirection,["Direction"]),
14
15 dataTransform("RestaurantAddress", "Address"),
16
17 initial(NumberOfGuest),initial(BookingDate), initial(BookingTime), initial(GuestName),
18 goals(["RestaurantName","BookingConfirmation","Direction")].
19
20 op_list([fromCoordinatesToCity,findFinestRestaurant,bookRestaurant,getDirection]).
```

Listing 3.1: FLUX generated query for the motivating example

With the above specification, the *State* of the FLUX query is assigned to Z0 (line 1). The FLUX query has initial states for example, initial(RestaurantType) and initial(Coordinates) (line 17) and a set of goal states in goals([RestaurantName, BookingReservation, Direction]) (line 18). The query contains 4 service operations (line 20): fromCoordinatesToCity, findFinestRestaurant, bookRestaurant and getDirection. Each operation provides a set of inputs and a set of outputs.

For instance, operation fromCoordinatesToCity defines its input Coordinates in op_inputs(fromCoordinatesToCity, [Coordinates]) (line 4) and its output City in op_outputs(FromCoordinatesToCity, [City]) (line 10). The query also specifies data transform between term RestaurantAddress and term Address in dataTransform(RestaurantAddress,Address) (line 15). Without it, the agent is not able to make a link between operation findFinestRestaurant and operation getDirection.

Whereas, the partial fluents for abstract plan structures are listed below:

```
Add_initail(X): OPERATION --> FLUENT
Add_goal(X): OPERATION --> FLUENT
Flow(X,Y): OPERATION x OPERATION --> FLUENT
Flow(X, if(guard(O1,RO1,V1),Y1),else(guard(O2,RO2,V2),Y2)):
OPERATION x PARAMETER x OPERAND x VALUE x OPERATION x
PARAMETER x OPERAND x VALUE x OPERATION --> FLUENT
```

To construct the sequencing and paralleling plans, `Add_initail(X)`, `Add_goal(X)` and `Flow(X,Y)` fluents are required, where `Add_initail(X)` is a fluent for an initial node of operation `X` (i.e., *add_initail(FromCoordinatesToCity)*), `Add_goal(X)` is a fluent for a goal node of operation `X` (i.e., *add_goal(GetDirection)*) and `Flow(X,Y)` is a fluent for a flow node with a pair of head operation `X` and tail operation `Y` (i.e., *flow(FromCoordinatesToCity,FindFinestRestaurant)*).

If fluent `Op_inputs(X,Ys)` is hold, this means all inputs Ys of operation X are matched to either initial parameters or output parameters of other operations. Thereafter, the agent adds the fluents either `Add_initial(X)` or `Flow(X,Y)` into the abstract plan.

To merge condition operations into the abstract plan, the service composition agent uses `Flow(X, if(guard(O1,RO1,V1),Y1), else(guard(O2, RO2,V2),Y2))` fluent. The `Flow(X,if(guard(O1,RO1,V1), Y1), else( guard(O2,RO2,V2), Y2))` is fluent for a flow with condition of two alternative outgoing paths. The first one is a path from operation `X` to operation `Y1` if the guard condition that parameter value `O1` with relational operation `RO1` equals to `V1` is true. The second path is from operation `X` to operation `Y2` if the guard condition that parameter value `O2` with relational operation `RO2` equals to `V2` is true. More fluents for the FLUX query and abstract plan structures are listed in Appendix A.

3.4.2 Service composition agent

The previous Subsection, we have described structures of the FLUX query and the abstract plan, which contains fluents of initial, goal states, service operations and elements of the plan respectively. However, the service composition agent needs to perform planning domain actions which includes precondition axiom and update axioms in order to execute the plan. Thus, in this subsection, two questions have been addressed. What are the planning domain actions the agent can perform? and what is the agent strategy to achieves the resulting plan?. Our agent basically performs adding initial nodes, flow nodes, flow with condition nodes and goal nodes into the abstract plans. We explain some planning domain actions the agent perform as follows:

Add_initial(X,Z1,Z2) is an action for the agent to add initial node of operation X into the abstract plan. Consider fragment code Listing 3.2, the agent checks its state in Z that there is none of head operation X of any flow(X,_) exists in tail operation of any flow(_,X) (line 1-5) and then updates its state from Z1 into Z2 by adding add_initial(X) fluent (line 7-11).

```
1  check_initial([],Z,Z).
2  check_initial([X|Op_list1],Z,Zn) :-
3          ((holds(flow(X,_),Z), not_holds_all(flow(_,X),Z)))
4                          -> (add_initial(Op,Z,Z1), check_initial(Op_list1,Z1,Zn))
5                          ;check_initial(Op_list1,Z,Zn).
6
7  add_initial(X,Z1,Z2)    :-
8          state_update(Z1,add_initial(X),Z2,[]).
9
10 state_update(Z1,add_initial(X),Z2,[]) :-
11         update(Z1,[add_initial(X)],[],Z2).
```

Listing 3.2: Agent performing adding initial node action

Add_goal(X,Z1,Z2) is an action for the agent to add goal node of operation X into the abstract plan. Consider fragment code in Listing 3.3, the agent checks its state in Z that there is an existence of a single goal G in a set of operation output Outputs and the operation X must not exist in any flow(X,_) (line 1-4). Thereafter, the agent updates its state from Z1 into Z2 by adding add_goal(X) fluent (line 6-10).

```
1  check_goal([],_,_,Z,Z).
```

```
2  check_goal([G|Gs],Outputs,X,Z,Zn) :-
3          (member(G,Outputs), not_holds_all(flow(X,_),Z))
4                  -> (add_goal(Z,X,Z1), check_goal(Gs,Outputs,X,Z1,Zn)).
5
6  add_goal(X,Z1,Z2)       :-
7          state_update(Z,add_goal(X),Z2,[]).
8
9  state_update(Z1,add_goal(X),Z2,[]) :-
10         update(Z1,[add_goal(X)],[],Z2).
```

LISTING 3.3: Agent performing adding goal node action

Another action is Add_flow(Z1,X,Y,Z2). This action is called when the agent wants to add flow node linking operation X and operation Y into the abstract plan. Consider fragment code in Listing 3.4, the agent checks its state in Z that there is an existence of Output of operation X in a set of Inputs of operation Y (line 2-5). Thereafter, the agent updates its state from Z1 into Z2 by adding add_flow(X,Y) fluent (line 7-11).

```
1  check(_,_,[],Z,Z).
2  check(Output,X,[Y|Op_list],Z,Zn)        :-
3          knows_val([Inputs],op_inputs(Opy,Inputs),Z),
4          ((member(Output,Inputs) -> (add_flow(Z,X,Y,Z1),
5                                      check(Output,X,Op_list,Z1,Zn))).
6
7  add_flow(Z,X,Y,Z1) :-
8          state_update(Z,add_flow(X,Y),Z1,[]).
9
10 state_update(Z,add_flow(X,Y),Z1,[]) :-
11         update(Z,[flow(X,Y)],[],Z1).
```

LISTING 3.4: Agent performing adding flow node action

Apart from the FLUX query and the planning domain actions, a behavior of FLUX planner is mandatory for the abstract service composition as it governs its actions to achieve the resulting plan.

Algorithm 3.5 is a part of a program for service orchestration planning agent. The objective of the program is to synthesize the abstract plan from initial knowledge of user's initial and goal states, a list of service operation and predefined user's constraints. To do so, the initialization of planning variables is performed that all FLUX query fluents are in state Z0 (line 2), two lists of operations are assigned in Op_list1 and Op_list2 (line 3), variable Gs keeps a set of goal fluents (line 4) and variable CList stores a list of user's constraints (line 5). Then the agent visits each

of a given operation in Op_list1 and systematically check matching an output of one operation in Op_list1 to an input of another operation in Op_list2 in situation `check_flow(Op_list1, Op_list2, Z0,Z1)` (line 6). After learning the matching, the agent creates a flow of two linked operations into a plan.

Next (line 10), the agent checks whether all goals Gs are hold among given operations Op_list1 or not. If the agent finds one operation output matches to a goal in Gs then the goal node of that operation is added into the plan.

```
1  main3 :-
2          init(Z0),
3          op_list(Op_list1),op_list(Op_list2),
4          knows_val([Gs],goals(Gs),Z0),
5          condition_list(CList),
6          (check_flows(Op_list1,Op_list2,Z0,Z1)
7                                  -> Fflow=true
8                                  ;
9                                  Fflow=false, Z0=Z1),
10         check_goals(Gs,Op_list1,Z1,Z2),
11         check_initial(Op_list1,Z2,Z3),
12         check_OR_split(CList,Z3,Z4),
13         check_OR_join(CList,Z4,Z5),
14         writeln(Z5).
```

Listing 3.5: Part of an agent program to plan service orchestration

The agent monitors which operation is a origin of the plan (line 11). The agent searches for the head operation in the generated flow fluents which do not match to tail operation of any other flow fluents. As the result, the initial node is added to the plan.

Lastly, the agent checks constraints user specified in CList. There are two kinds of condition requests: request with XOR split and request with XOR join. The former (in line 12) determines a split of one flow operation into two other operation flows if their conditions are hold. If agent is able to search for operations satisfied for the request with XOR split then the flow with XOR split node is created. The latter (in line 13) specifies a join one flow operation from other operation flows. If agent is able to search for operations satisfied for the request with XOR join then the flow with XOR join node is created into the plan. As the end, the resulting plan fulfills the specified user goals and also supports sequencing, paralleling and conditioning among service operations. The complete program for service orchestration planning agent is shown in Appendix B.

As the result of the FLUX planner process subjective to the motivating example, the service composition agent generates the following fluents:

```
add_initial(fromCoordinatesToCity),
flow(fromCoordinatesToCity,findFinestRestaurant),
flow(findFinestRestaurant,bookRestaurant),
flow(findFinestRestaurant,getDirection),
add_goal(bookRestaurant),
add_goal(getDirection).
```

The agent generates a plan as a composition of operations for Alice's query that she wants to get a booking reservation and a direction to the restaurant. In other words in Figure 3.5, given two goals bookingConfirmation and direction and and a list of service operations to the agent, the composite process starts calling operations fromCoordinatesToCity and then findFinestRestaurant. After the operations bookRestaurant and getDirection. Finally, the agent generates the composition of operations that is satisfied the goals bookingConfirmation and direction.

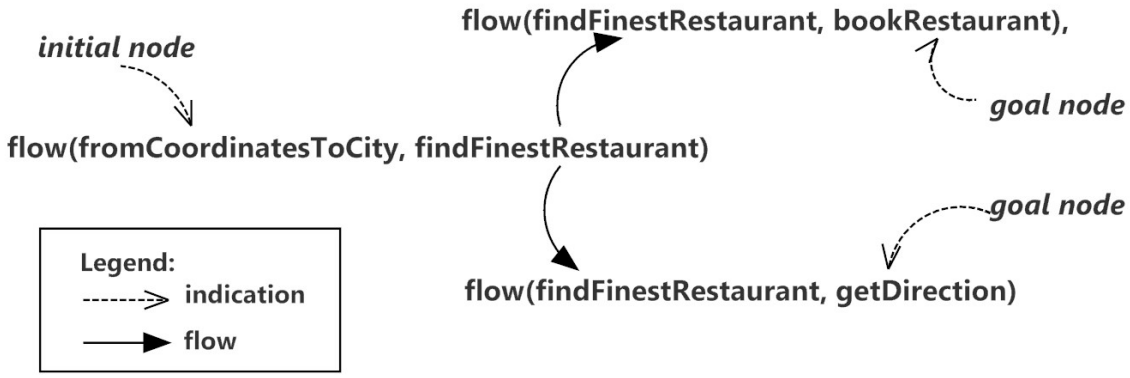


FIGURE 3.5: The abstract service composition for Alice's query

3.5 Existing approaches

In this section, we compare our proposed abstract service composition system to the existing approaches in term of service inputs, resulted plan outputs and automation level. We found three existing works, which model planning domain including knowledge of initial and goal states and agent actions in propositional logic approaches such as situation calculus and fluent calculus and develop

Systems	**Services**	**Plan support**	**Automation level**
Viorica et al	Atomic	sequence	Auto
Golog	Atomic	sequence, condition, parallel	Semi-auto
WSSL	Atomic and composite	sequence, condition, parallel	Semi-auto
Our proposal	Atomic	sequence, condition, parallel	Auto

TABLE 3.3: Comparison among planner systems

planning reasoner to solve its problems using constraints programming languages [LRL+97, CSHG09, BP14].

The GOLOG has proposal to solve automatic service composition. The authors modeled problems and actions in logic using situation calculus. They use ontology for facilitating service discovery for atomic service. However, they use user template to control the workflow of the plan [LRL+97]. This makes their approach semi-automated whereas we propose automated service composition.

Mapping between OWL-S service operations and FLUX actions is proposed in [CSHG09]. FLUX constraint programming is used to conduct a resulting plan. However, the generated plan shown has only a sequence of operations where our resulting plan can support sequencing, conditioning and paralleling among operations.

Another relevant work is WSSL[2]. This work has been proposed recently the service description specification using fluent calculus. and their services cover both simple and complex service [BP14]. However, for the service composition, the authors offered semi-auto method to generate a workflow using behavior logic template. The below Table 3.3 is shown a summary of the comparison among these relevant systems.

3.6 Summary

In this chapter we present the Abstraction Service Composition which aims to automatically compose services on the fly. In doing so, we adopt AI-planning

[2]Web Service Specification Language

technique to solve automated service composition problems. Given a list of operations and user goals, the AI-planner build up an abstract plan, which is a sequence of operations. Therefore, we have called orchestration, workflow and abstract plan in this dissertation interchangeably. We have proposed constraints programming system being our operations composer. The composer is capable to generate the abstract plan, which has sequencing, conditioning and parallelizing controls among operations.

Chapter 4

Composition Platform Generation

Contents

Abstract. Our composition platform layer to valid semantic BPMN model of the composite platform generation system, one of our contributions, is presented in this Chapter. The objective of the system is to analyze and check the properties of derived BPMN model from the workflow of abstract service composition system. Therefore, a model representing BPMN process in Prolog language is proposed in this chapter. Two processes of BPMN Transformer and BPMN Validation respectively are presented and detailed.

This chapter discusses the Composite Platform Generation, which is the process present in Composite Platform layer of the Service Composition and Execution system (see Figure 4.1).

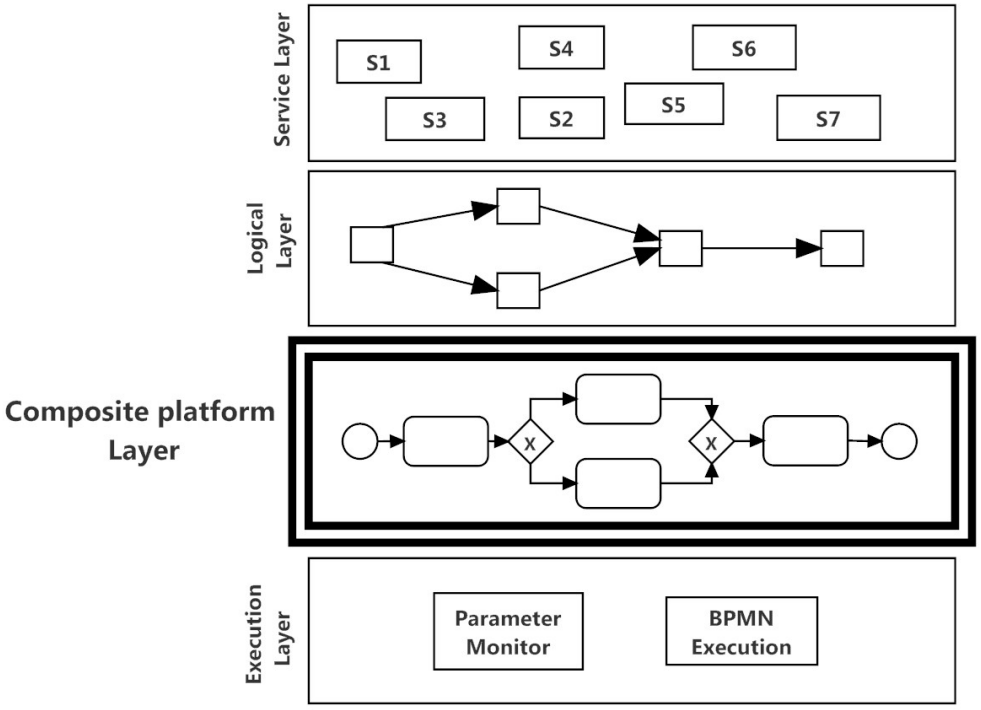


FIGURE 4.1: Proposed multi-layers of the Service Composition and Execution system highlighted at Composition Platform layer

The Composite Platform generation process handles transforming the abstract plan, which is derived from the Abstract Service Composition in the Logical layer, to BPMN model in Prolog language and analyzing the BPMN model whether it is well-formed BPMN from the BPMN specification[OMG] and well-defined BPMN from correctness requirements.

Therefore, we start this chapter with an architecture of the Composition Platform Generation process in Section 4.1. As we selected BPMN as our composition platform, we describe in details abstract plan to BPMN semantic in Section 4.2. The main components of the Composition platform generation process: the BPMN Transformation and the BPMN Validation are explained in Sections 4.3 and 4.4 respectively.

4.1 Architecture

A preliminary aim of the Composite Platform Generation process is to obtain an executable composite service, given an input of the abstract plan from the Abstract Service Composition process (see in Chapter 3). As the abstract plan supports

only a control among abstract service operations, an execution of the composite plan is not be possible.

A solution considered is to create a workflow application. A workflow application is an information system that supports controlling the execution of complex application processes in a variety of domains, including the traditional business domain [WGHS99]. While a workflow itself is a collection of related, structured activities or tasks that produce a specific service. To develop the workflow application, any specialized workflow languages (i.e., WS-BPEL and BPMN) or general-purpose languages (i.e., XPDL[1] and YAWL[2]) can be used for workflow definition.

Among these languages, BPMN was selected for being our workflow composition platform. In other words, BPMN model shall represent a workflow of the abstract plan. Therefore, we need a mechanism for mapping the abstract plan into the BPMN model.

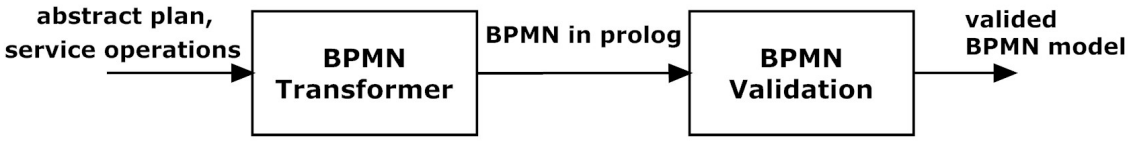


FIGURE 4.2: BPMN generation process

The Figure 4.2 illustrates the architecture of Composite Platform (or business process) Generation. The process starts from the BPMN transformer that converts the abstract plan, which is consisted of a sequence of operation fluents, to BPMN model in Prolog language. Next, the BPMN model is analyzed and verified in the BPMN Validation. The valid BPMN model as a result will be implemented and executed in the experiment phase later on.

[1]The XML Process Definition Language (XPDL) is a format standardized by the Workflow Management Coalition (WfMC) to interchange business process definitions between different workflow products.

[2]YAWL (Yet Another Workflow Language) is a workflow language based on workflow patterns.

4.2 Abstract plan to BPMN semantics

A BPMN is a standard notation maintained by OMG[3] for modeling business processes. Its goal is to provide a notation of business specification that is understandable by all business stake holders (i.e. business analysts, software developer and business people), mainly at the level of domain analysis and high-level systems design [OMG]. The BPMN is widely-used in the early stages of systems life cycle. According to the OMG, 72 implementations of the BPMN are reported for known businesses [OMG]. Moreover, open sourced software companies (i.e., Activiti[4], BonitaSoft[5] and Yaoqiang BPMN Edior[6]) dramatically compete among each others to offer varied solutions to edit and run BPMN models.

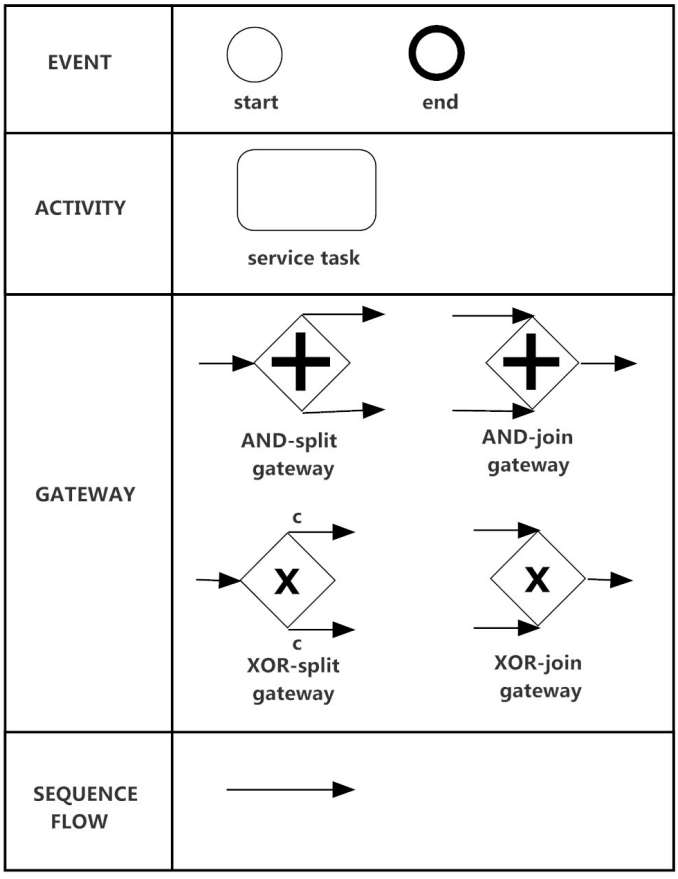


FIGURE 4.3: BPMN notation related to the proposed of our work [OMG]

BPMN is comprised of an abstract of workflow components. However, this dissertation focuses on a control-flow perspective of BPMN. Therefore, the subset of the

[3]http://www.omg.org/
[4]http://activiti.org/
[5]http://www.bonitasoft.com/
[6]http://bpmn.sourceforge.net/

notation that handles the order of activities are allowed to occur. It does not handle its non-functional features (i.e., artifacts and association) and organizational modeling features (i.e., lanes and pools).

Figure 4.3 shows an overview of a set of graphical BPMN elements related to the proposes of our work. For the event elements, only **start event** and **end event** are taken into consideration. **Service tasks** are main knowledge of processing elements; each task is a finite process with a set of inputs and a set of outputs. Two gateways: split and join control a workflow. **Split gateways** present when branching of the workflow takes place; two disjoint subtypes of splits are **AND-split gateway** and **XOR-split gateway**. AND-split allows a single flow to be split into two or more branches which can execute tasks concurrently while XOR-split allows a flow to be split into two or more flows when the incoming flow is enabled, the gateway is passed to one of the outgoing flows based on a specified condition that can select one of the outgoing flows. **Join gateways** happen when two or more paths meet; two further disjoint subtypes of merge modes are considered: **AND-join gateway** and **AND-split gateway**. AND-join allows two or more parallel flows to be joined into a single subsequent flow when all input flows have been enabled while AND-split allows a single flow to be split into two or more branches which can execute tasks concurrently. Lastly, a **sequence flow** is used to link two entities of event, activity or gateway in a process diagram and specify a control flow relation. For further details of BPMN specification, see Appendix C.

For BPMN transformation purpose, we map from logical analysis of BPMN component to their logical models, properties and representation in Prolog. The following table 4.1 lists BPMN elements along with their mapped semantic fluents into consideration:

BPMN elements	**BPMN semantics**
Start event	node(start)
End event	node(end)
Service task	task(Name,Inputs,Outputs)
Sequence flow	flow(HOperation,TOperation)
XOR-split gateway	gateway(xorS)
XOR-join gateway	gateway(xorJ)
AND-split gateway	gateway(andS)
, AND-join gateway	gateway(andJ)

TABLE 4.1: Mapping between BPMN elements and BPMN semantics

The BPMN elements are mapped into the BPMN semantics according to the element type. For example, Start event is mapped to node(start) and so on. However, Service task and Sequence flow BPMN element require more informations for semantic mapping. A semantic of Service task needs informations of task name, inputs and outputs for defining task(Name, Inputs, Outputs), where Name, Inputs and Outputs are variable names. The same principle applies to Sequence flow that it needs a workflow information of head operation linking to tail operation for flow(HOperation,TOperation), where HOperation and TOperation are operation names.

4.3 BPMN Transformer

The BPMN transformer is a process for mapping the abstract plan consisting of a sequence of the fluents into semantic BPMN model in the declaration of a formal language. The reason why we transform the abtract plan into the formal language is that a graphical notation of BPMN elements binds information of data passing, data transformation and routing condition from BPMN workflow specification. This shall be hard to fix defects when they occur in BPMN model.

We selected Prolog as a formal language for modeling BPMN. The static analysis towards a logical model for BPMN diagram will be discussed later in Section 4.4. Therefore, the resulting transformed model shall represent semantic workflow specification in Prolog including data passing, data transforming and routing condition.

The following Table 4.2 shows mapping rules between particular fluents occurred in the abstract plan and semantic BPMN we defined in Table 4.1. We have classified the fluents into four groups: *initial group, flow group, goal group* and *data group.*

The *initial group* contains add_initial(Op) fluents, where Op is a single operation. Mapping these initial fluents into BPMN could create two possible situations. The first situation happens when the transformer detects only one initial fluent in the abstract plan. The transformer creates node(start) and flow(start,Op), linking between start event and operation Op, into BPMN model. While the second situation occurs when there are more than one initial fluents in the plan. This means operations derived from initial fluents can start a process at the same time. A converted BPMN has one AND-split gateway to combine these initial operations. For example, add_initial(Op1), add_initial(Op2) is mapped to BPMN model, which is consisted of node(start), gateway(andS), flow(start,andS), flow(andS,Op1), flow(andS,Op2).

The *flow group* contains flow(_,_), flow(_,if(guard(_,_,_,_),_),else(guard(_,_,_,_),_)) and flow(if([_,_],_)) fluents. These flow fluents can be mapped into a sequence, parallel and condition flows in BPMN model. For the *sequence flow*, the transformer does not change flow(_,_) fluent. For instance, flow(Op1,Op2), where Op1 and Op2 are instants of Task Op1 and Task Op2, stays remain in BPMN since it describes a control flow from task object to another task object in the same way as a flow dose in BPMN. While creating *parallel flow* in BPMN, the transformer checks all tasks in the head position of flow fluent. i.e., OP1 in flow(Op1,_) fluents whether Op1 exists in any head position of other flow(Op1,_) fluents. If these flow fluents exist, the transformer convert them to one AND-split gateway, relevant control flows and relevant tasks. For example, flow(Op2,Op3), flow(Op2,Op4) is mapped to gatway(andS), flow(Op2,andS), flow(andS,Op3), flow(andS,Op4). This checking parallel rule also is applied for AND-join gateway that the transformer monitors the tail position of flow fluent. i.e., Op1 in flow(_,Op1) fluent with others flow(_,Op1) fluents instead. For the *condition flow*, the transformer checks the abstract plan for flow(_,if(guard(_,_,_,_),_), else(guard(_,_,_,_),_)) fluent and flow(if([_,_],_)). If the former fluent is detected, the transformer creates one XOR-split gateway, one control flow and two control flows with condition into BPMN model. For example, flow(Op1,if(guard(O1,RO1,V1),Op2),else(guard(O2,RO2,V2),Op3) is mapped to gatway(xorS), flow(Op1,xorS), flow(xorS,Op2,guard(O1,RO1,V1)),

Group	Fluents in Abstract plan	BPMN
Initial	*initial* add _initial(Op1)	node(start) flow(start, Op1)
	initial with parallel add _initial(Op1) add _initial(Op2)	node(start) gateway(andS) flow(start,andS) flow(andS,Op1) flow(andS,Op2)
Flow	*flow with sequence* flow(Op1,Op2)	flow(Op1,Op2)
	flow with AND Split parallel flow(Op2,Op3) flow(Op2,Op4)	gateway(andS) flow(Op2,andS) flow(andS,Op3) flow(andS,Op4)
	flow with AND Join parallel flow(Op3,Op5) flow(Op4,Op5)	gateway(andJ) flow(Op3,andJ) flow(Op4,andJ) flow(andJ,Op5)
	flow with XOR Split condition flow(Op1, if(O1,S1,V1,Op2), else(O2,S2,V2,Op3)	gateway(xorS) flow(Op1, xorS) flow(xorS,Op2,guard(O1,S1,V1)) flow(xorS,Op3,guard(O2,S2,V2))
	flow with XOR Join condition flow(if([Op2,Op3], Op4)	gateway(xorJ) flow(Op2, xorJ) flow(OP3,xorJ) flow(xorJ,OP4)
End	*goal* add_goal(Op4) add_goal(Op5)	node(end1) node(end2) flow(Op4, end1) flow(Op5, end2)
Data	*data passing* op_inputs(Op,[Inputs]) op_outputs(Op,[Outputs])	task(Op, [inputs], [outputs])
	data transforming dataTransform(X,X1)	dataTransform(X,X1)

TABLE 4.2: Mapping between the abstract plan and BPMN workflow

flow(xorS,Op3,guard(O2,RO2,V2)). While the flow with XOR-join gateway is created if the latter fluent is captured. For instance, flow(if[Op2,Op3],Op4) is mapped

to gatway(xorJ), flow(Op2,xorJ), flow(xorJ,Op3), flow(xorJ,Op4).

The *goal group* contains add_goal(Op) fluents, where Op is a single operation. We follow BPMN specification that a workflow in BPMN model may have more than one end events. Therefore, the transformer creates goal nodes up to number of distinct add_goal(_) fluents found. For example, add_goal(Op4), add_(Op5) is mapped to node(end1), node(end2), flow(Op4,end1), flow(Op5,end2).

Lastly, the *data group* contains data passing and data transforming fluents. The data passing describes both the task Op creation and its data passing of inputs and outputs. To do so, the transformer searches for service operations used in the abstract plan. For each operation Op, the transformer combines op_inputs(Op,[inputs]) and op_outputs(Op,[outputs]) fluents and map them to task(Op,[inputs],[outputs]) fluent, referring to Task Op containing a set of its inputs and its output. While dataTransform(X,X1) fluent, capturing a change from a data form X into another form X1, in the abstract plan stays remain in the semantic BPMN model.

4.3.1 Example of BPMN model

This subsection illustrates an example of BPMN Transform process mapping between the abstract plan and the semantic BPMN model. The example scenario is taken from Section 3.2. Before the mapping, the abstract plan is consisted of:

- add_initial(fromCoordinatesToCity), referring operation fromCoordinatesToCity is added as initial to the plan,
- add_goal(bookRestaurant) and add_goal(getDirection), referring that bookRestaurant and getDirection are operations holding user's goals in the plan,
- flow(fromCoordinatesToCity,findFinestRestaurant), flow(findFinestRestaurant, bookRestaurant) and flow(findFinestRestaurant, getDirection), each flow fluent referring to a control flow from the 1st operation to the 2nd operation.
- dataTransform(RestaurantAddress,Address), referring to data transforming from parameter RestaurantAddress to parameter Address. Without this dataTransform fluent, flow(findFinestRestaurant,getDirection) fluent is not existed.

- op_inputs(fromCoordinatesToCity,[Coordinates]), op_inputs(bookRestaurant, [RestaurantName,BookingDate, BookingTime,GuestName,GuestTelephone]), op_inputs(findFinestRestaurant,[City]) and op_inputs(getDirection, [Coordinates,Address]), each referring to a set of inputs of the operation.

- op_outputs(fromCoordinatesToCity, [City]), op_outputs(findFinestRestaurant, [RestaurantName,RestaurantAddress]), op_outputs(bookRestaurant, [BookingConfirmation]) and op_outputs(getDirection, [Direction]), each referring to a set of outputs of the operation.

The BPMN transformer converts the above mentioned fluents into the semantic BPMN model (see in Tables 4.3 and 4.4). We group the fluents to define BPMN elements:

1. Start node and its flow
   As the transformer detect only a single add_initial(fromCoordinatesToCity) fluent in the plan, the transformer reasons for fluents subjective to operation FromCoordinatesToCity and creates node(start), task(fromCoordinatesToCity,[coordinates],[city]) and flow(start, fromCoordinatesToCity).

2. Sequence flow between two operations
   The sequence flow between two operations happens in the BPMN model when the transformer detects non parallel gateway from that two operations. Therefore, the transformer maps flow(fromCoordinatesToCity,findFinestRestaurant) and its findFinestRestaurant relevant fluents into flow(fromCoordinatesToCity,findFinestRestaurant) and task(findFinestRestaurant,[coordinates],[name(restaurant),address(restaurant)]) in the semantic BPMN elements.

3. Gateway and its flows
   As the transformer detects parallel operation findFinestRestaurant among these flow(findFinestRestaurant,bookRestaurant) and flow(findFinestRestaurant,getDirection) fluents, this indicates an occurrence of AND-split gateway. Therefore, the mapped BPMN elements are gateway(andS1), flow(findFinestRestaurant,andS1), flow(andS1,bookRestaurant), flow(andS1,getDirection), task(bookRestaurant,[name(restaurant),date,time,

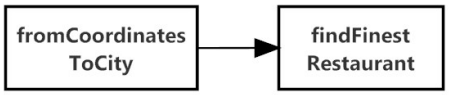
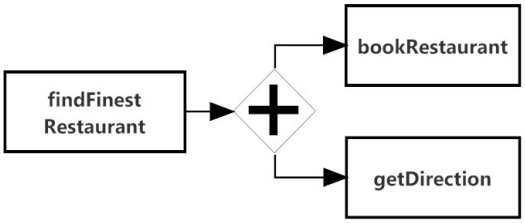
Abstract plan in Fluent calculus	**Semantic BPMN model and diagram**
add _initial(fromCoordinatesToCity) op_inputs(fromCoordinatesToCity, [Coordinates]) op_outputs(fromCoordinatesToCity, [City])	node(start) task(fromCoordinatesToCity, [Coordinates], [City]) flow(start, fromCoordinatesToCity) 
flow(fromCoordinatesToCity, findFinestRestaurant) op_inputs(findFinestRestaurant, [Coordinates]) op_outputs(findFinestRestaurant, [RestaurantName, RestaurantAddress])	flow(fromCoordinatesToCity, findFinestRestaurant) task(findFinestRestaurant, [City], [RestaurantName,RestaurantAddress]) 
flow(findFinestRestaurant, bookRestaurant) flow(findFinestRestaurant, getDirection) dataTransform (RestaurantAddress,address) op_inputs(bookRestaurant, [RestaurantName,BooingDate, BookingTime,GuestTelephone]) op_outputs(bookRestaurant, [BookingConfirmation]) op_inputs(getDirection, [Coordinates,Address]) op_outputs(getDirection, [Direction])	gateway(andS1) flow(findFinestRestaurant, andS1) flow(andS1, bookRestaurant) flow(andS1, getDirection) task(bookRestaurant,[RestaurantName, BookingDate,BookingTime, GuestTelephone],[BookingConfirmation]) task(getDirection, [Coordinates,Address], [Direction]) dataTransform(RestaurantAddress,Address) 

Table 4.3: Mapping service operations example between problem domain and planning domain

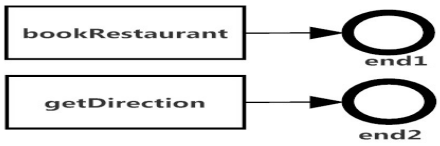
Abstract plan in Fluent calculus	**Semantic BPMN model and diagram**
add_goal(bookRestaurant) add_goal(getDirection)	node(end1) node(end2) flow(bookRestaurant,end1) flow(getDirection,end2) 

TABLE 4.4: Mapping service operations example between problem domain and planning domain (cont.)

telephone], [bookingConfirmation]), task(getDirection,[coordinates,address], [direction]) and dataTransform(address(restaurant),address).

4. End nodes and their flow.
   As the transformer captures two add_goal(bookRestaurant) and add_goal(getDirection), the semantic BPMN elements: node(end1), node(end2), flow(bookRestaurant,end1) and flow(getDirection,end2) are created in the BPMN model.

The Figure 4.4 illustrates an aggregation of the semantic BPMN elements in Tables 4.3 and 4.4. The BPMN model can be read that a process starts from fromCoordinatesToCity operation followed by findFinestRestaurant operation. Later, the findFinestRestaurant operation is split into bookRestaurant and getDirection operations and the process is then ended.

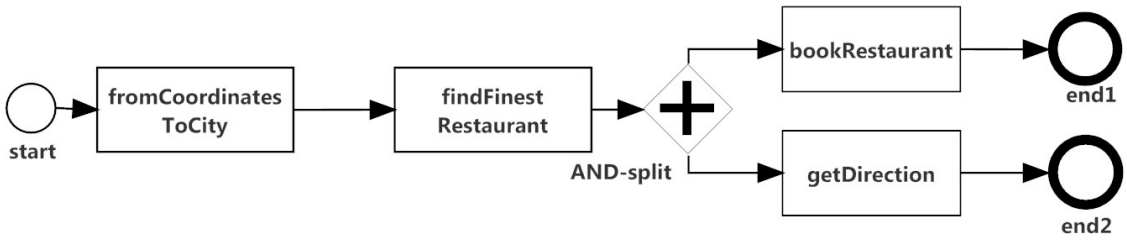


FIGURE 4.4: The example of BPMN model (Alice's process)

4.4 BPMN Validation

The BPMN Validation is a process to analyze and verify BPMN models using AI approach. To do so, we formulate checking rules against well-formed BPMN to assure the BPMN model specification. The rules also reveal guard condition expression on the semantic BPMN model.

4.4.1 Well-formed BPMN process

We show some useful Prolog queries for finding properties of the business process model (see more queries in Appendix C.3). For this purpose we use the rules defined in the specification of well-formed BPMN processes [OMG], see Appendix C.2.

- `rule1(Z) :-`
  `holds(flow(start,_),Z).`

  This rule verifies that a start event must be a source of a sequence. Given a semantic BPMN model in State Z, the validation agent checks whether there is a flow(start,_) fluent in Z under predicate holds(flow(start,_),Z). If a result returns *true*, it refers that there is a start event being a source of a sequence in the BPMN model.

- `rule2(Z) :-`
  `\+ (knows_val([End],node(End),Z), End \= start,`
  `knows_val([Op],flow(Op,End),Z), holds(flow(start, Op),Z)).`

  This rule verifies that there must not exist a connection between a start and an end event. Given a semantic BPMN model in State Z, the validation agent retrieves variable End from node(End) fluent in Z under predicate know_val([End],node(End),Z) and check the End value is not equal to start node. The agent then retrieve variable Op in the flow fluent having the End being an end of a sequence in Z under predicate know_val([Op],flow(Op,End),Z) and lastly the agent checks whether there is a flow(start,Op) fluent in Z under predicate holds(flow(start,Op),Z). With logical negation \+ in Prolog, the agent repeats the same procedure on other end node fluents. The \+ will return *true* only if every loops return *false*.

This refers that there is none of a connection between a start and an end event.

4.4.2 Well-defined BPMN process

The above mentioned rules only verify BPMN well-formed model. However, a well-formed model does not assure that any input knowledge the process can be executed leading to expected goals. Therefore, we have following auxiliary functions (line 1-9) aiming to check all possible process.

```
1  travel(S,E,[S|P],[L|_],GE,Z,Z1) :-
2  holds(flow(E,L),Z), update(Z,[flowPath([A|P],GE)],[],Z1).
3
4  travel(S,E,Visited,P,GE,Z,Z1)   :-
5  (holds(flow(C,E),Z) -> ((C\==S;C==start), not(member(C,Visited)),
6                          travel(S,C,[C|Visited],Visited,Cond1,Z,Z1))
7                       ;
8   holds(flow(C,E,GE2),Z,Z1) -> ((C\==S;C==start), not(member(C,Visited)),
9                          travel(S,C,[C|Visited],Visited,[GE2|GE],Z,Z1))).
```

After running the above functions, resulting model contains all possible paths, each path holding all visited nodes and guard conditions. To do so, the validation agent calls travel(S,E,[S|P],[L|_],GE,Z,Z1) action (line 1), where S = start node, E = end node, [S|P] = a set of visited nodes (start node (S) plus a sequence of nodes (P)), [L|_] = a set of visited nodes (operation node (L) next to the start node), GE = guard condition, Z = current state, Z1 = new state. The agent checks (line 2) whether there is a flow fluent having variable E of start node in a head position and a derived operation node L from another travel predicate (line 4-9) at tail position in state Z or not. With the travel predicate (line 4-9), the agent knows operation node C from holds(flow(C,E),Z) and checks a condition whether C /= start node S or C = start node. If this condition passes, C is checked that it is not member of Visited a set of visited nodes. Then operation node C is add into the visited path in [C|Visited]. The agent continues adding nodes into the visited path until the flow(E,L) is found in Z. If the semantic BPMN model contains flows with guard conditions, the validation agent will add a set of guard conditions into GE (line 8-9). Finally, the agent adds all possible paths [A|P] and relevant guard conditions GE into flowPath([A|P],GE) fluent under action update(Z,[flowPath([A|P],GE)],[],Z1).

This checking assures resulted well-defined model since the flow fluents, derived from FLUX planner, were processed using links detected matched parameters from two service operations, their data transformation, and correct controls defined by the branching/merging condition assigned to the links (see in Section 3.4, Page 59).

4.4.3 Related work

There are many researches based on BPMN formalization and verification. They migrate from logical analysis of BPMN component to their logical models, properties and representation in formalization languages such as π-Calculus [PW06], Petri nets [DDO08], YAWL [DDDGB08], Prolog [LP14] and Maude [ESB14].

Among these formalization languages, we selected AI approach to formally analyse our BPMN model. To the best of our knowledge, apart from [And10, LP14], there are no other related works defining BPMN model in Prolog. The former used Prolog modeling BPMN model for simulation purpose [And10]. The simulation of business process to assess the cost and time of running a process and to identify potential problems with resources. He claimed that his result is the same as the result from specialized commercial tools like IBM WebSphere Business Modeler[7].

The latter performed analysis against well-formed and well-defined BPMN process [LP14]. Their aim is to verify BPMN model in the declarative Prolog language on specification of correct components and correct data flow. Compare to our work, we also defined formal semantics of diagram components and workflow operation in Prolog language, however we added guard condition expression evaluation to check whether the solution holding the guard condition expression defined from correctness requirements.

4.5 Summary

This chapter talks about business process generation process. Since business process is the standard method to create the workflow application, which is one of our ultimate goals. Thus, we propose the mapping from the abstract plan, derived from the FLUX planner in the abstract service composition process, to semantic

[7]http://www-03.ibm.com/software/products/en/modeler-advanced/

BPMN model in the declarative Prolog language. Addition, we propose AI approach to formal analysis of the BPMN model. The analysis checking reveals the notation of well-formed and well-defined BPMN model.

Chapter 5

Implementation

Contents

Abstract. Implementation of our system for a proof of concept is presented in this Chapter. It covers the abstract service composition system in Chapter 3, the composite platform generation system in Chapter 4 and the parameter monitor and BPMN execution components, which are found in the execution layer. We present meta-model used to predefined objects of our proposed system. Finally, we show the experiments and results of our setup scenarios from Chapter 1.

This chapter talks about implementation of Service Composition and Execution system. The objectives of the system are (1) to generate correct and abstract BPMN workflow from the user's goals in propositional logic and the service operation references to aliases used to fulfill the propositional logic and (2) to implement and run executable BPMN model from the valid abstract BPMN model and users'

input parameters. For the first objective, the implementation in Section 5.2 covers FLUXQuery Transformation and FLUX Planner, which are introduced in Chapter 3, BPMN Transformation and BPMN Validation, which are introduced in Chapter 4. While the second objective, we discuss is in the implementation of Parameter Monitor and BPMN Execution. However, we discuss meta-model used for pre-defined objects of our approach problems in Section 5.1. Finally, a simulation of running examples from Chapter 1 is discussed in Section 5.3.

5.1 Models

The Service Composition and Execution system is comprised of various different domains such as user requirement, planning, web service and BPMN. Therefore, we have identified a set of 5 domain-specific models that together encapsulate the required range of process operations.

- **Requirement model.** A Requirement model, shown in Figure 5.1, represents an input requirement with a set of Initials. Generally, user provides the system informations of user's query, profile and context. These informations are preprocessed by the User query management; user's query is expressed in terms of goal and initial states, flow conditions and data transformation of certain parameters; user's profile and context are expressed in term of initial states. These informations are stored in Requirements. Since the system helps the user to enter values of input process parameters, Initials keeps information as expressed as the triple $< paraname, datatype, value >$.

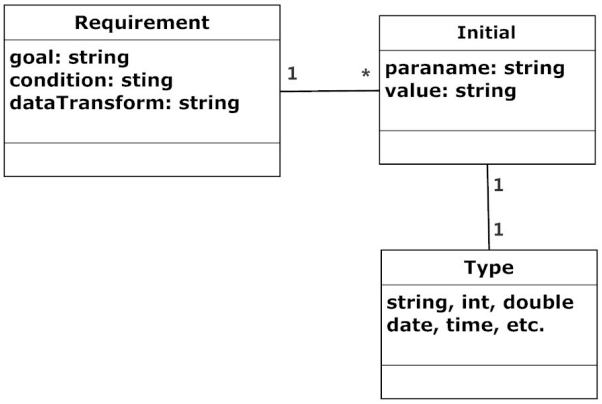


FIGURE 5.1: Requirement model

- **Service operation model.** To composition of service operations, as illustrated in Figure 5.2, we define a Service operation model. We support *wsdl* services that could be abstract into an Operation containing a set of Inputs and a set of Outputs. For an issue of service execution, we implement web service class called during process execution using org.activiti.engine.delegate.JavaDelegate interface. Under this interface, required business logic is implemented in the execute(DelegateExecution) method.

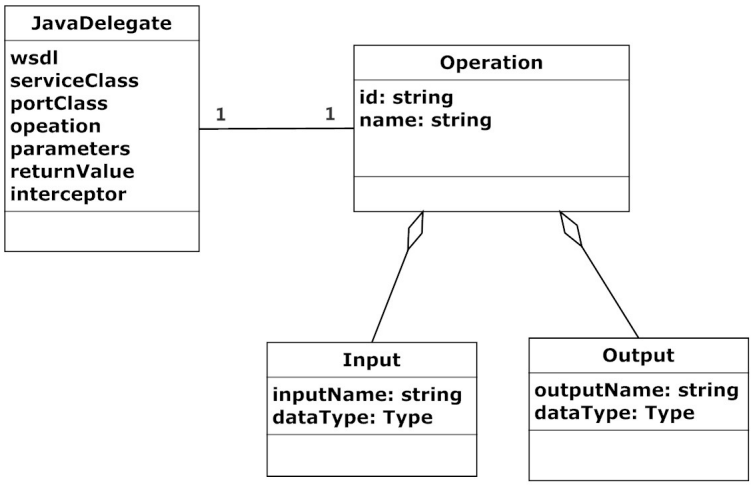


FIGURE 5.2: Service operation model

- **FLUX planner model.** The structure of our FLUX, explained in Section 3.1.2, is consisted of (1) encoding of goal knowledge state, service operation candidates and flow conditions, which are expressed as FLUXQuery, (2) domain axioms including action precondition axiom and update axiom and FLUX rules and constraints, which are expressed in agentFLUX and libraryFLUX respectively and (3) a high level of the agent's behavior expressed in programFLUX. Therefore, we implement these FLUX elements as FLUX planner model illustrated in Figure 5.3. An abstract plan resulted of *FLUX Planner* is kept in abstractPlanFile.

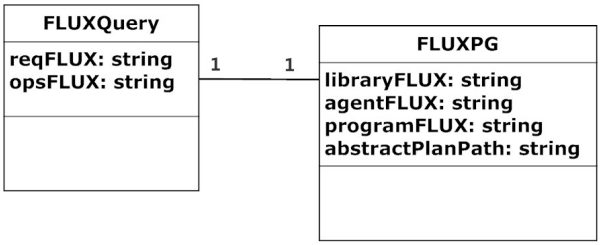


FIGURE 5.3: FLUX planner model

- **BPMN prolog model.** A BPMN prolog model, shown in Figure 5.4, represents two states of BPMN Prolog model object. They are states before and after validation. The BPMN state after validation reveals whether its model is valid against minimal rules of BPMN specification (see in Section 4.2) as expressed in isPropertiesValid and is correctness of user guard condition as expressed in isGuardValid.

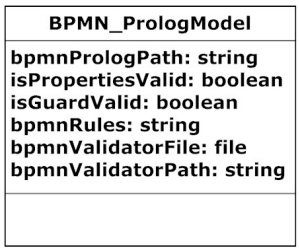


FIGURE 5.4: BPMN Prolog model

- **BPMN model.** This dissertation focuses on a control-flow perspective of BPMN. Therefore, the subset of the notation that handles the order of activities (Events, Flows, Gateways and Tasks) are allowed to occur.

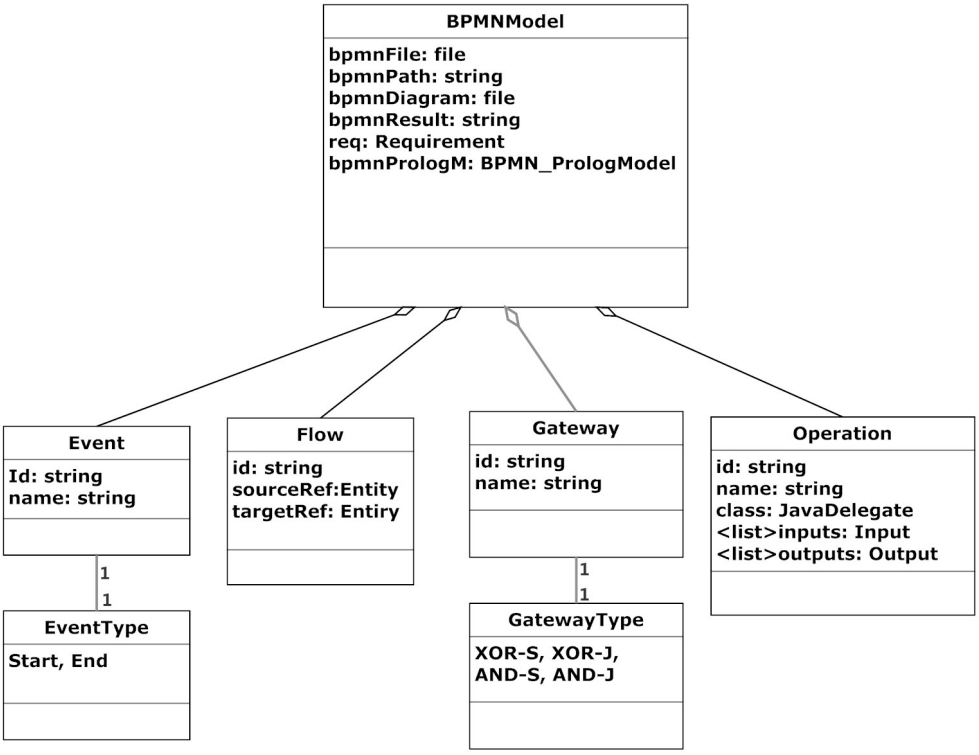


FIGURE 5.5: BPMN model

It does not handle its non-functional features (i.e., artifacts and association) and organizational modeling features (i.e., lanes and pools). Therefore,

we implement this BPMN model, as illustrated in Figure 5.5. Besides the BPMN elements, we keep other informations such as req.Initials as process input paramaters and bpmnResult as process output.

5.2 Implementation

Figure 5.6 illustrates the system design and interaction of the main components of the Service Composition and Execution system. We implement component class using our defined interfaces, expressed in Figure 5.7, in JAVA platform. The whole system operation can be summarized as follows. First a Requirement object

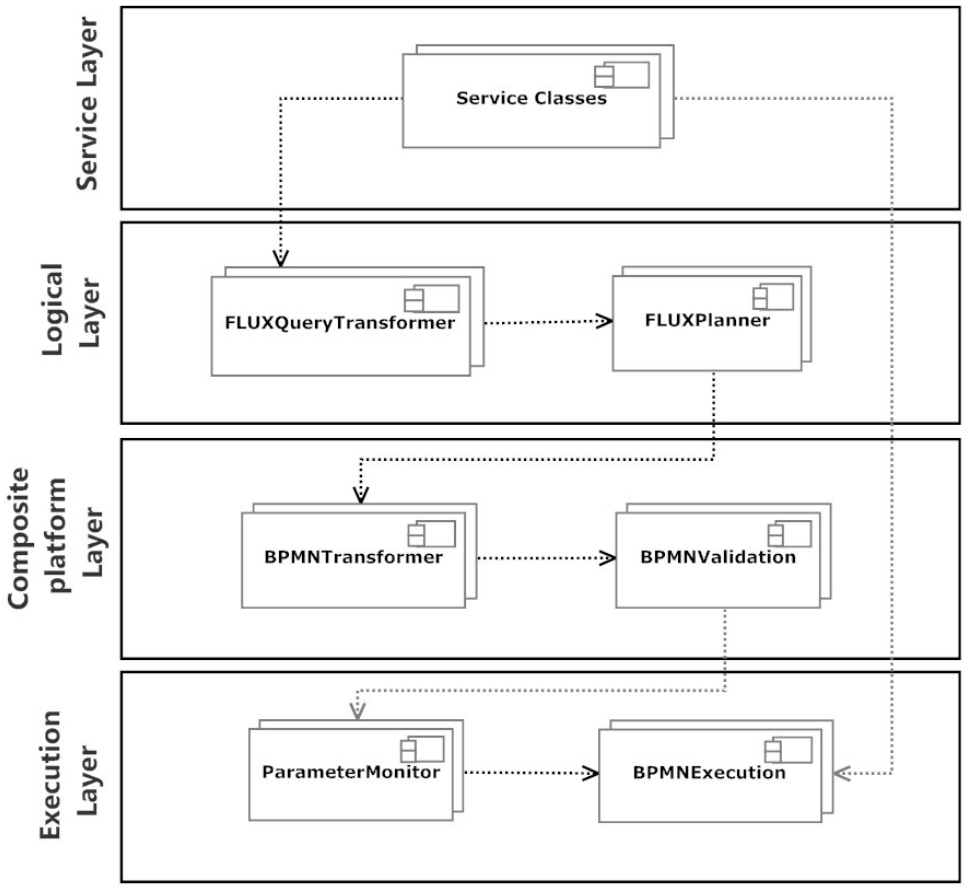


FIGURE 5.6: Composite platform layer on service composition and execution system

and a set of Operation objects are being transformed using operations in *FLUXQueryTransformer* component to obtain FLUXQuery object. Next *FLUXPlanner* component is triggered, which FLUX planning program is generated. To compile and run the program, we implement an object of *com.parcltechnologies.eclipse*,

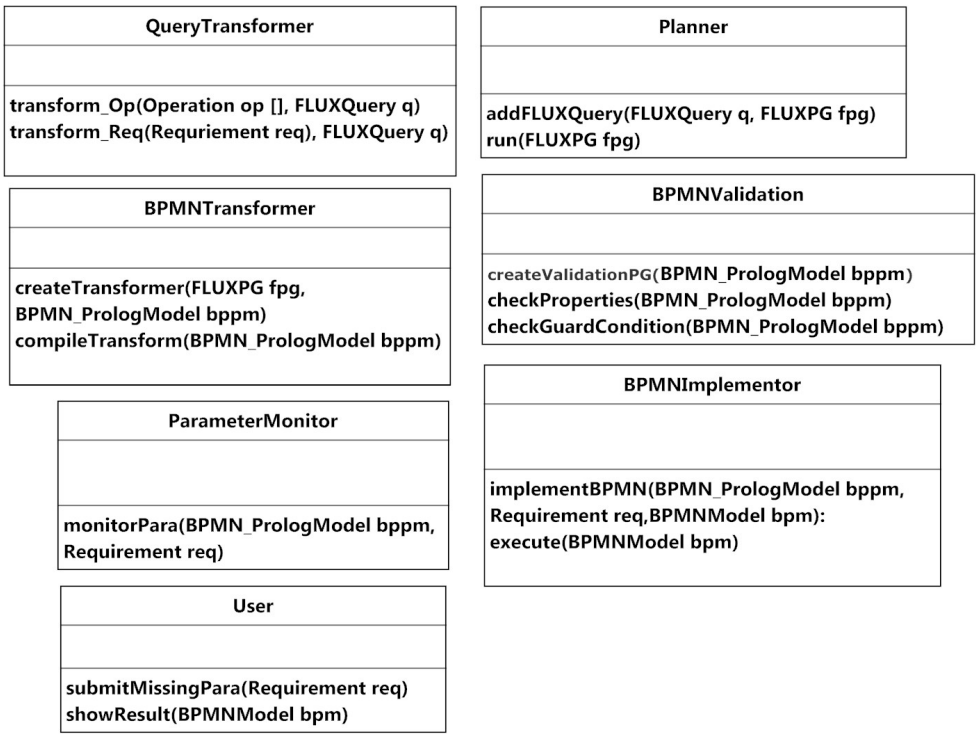


FIGURE 5.7: Composite platform layer on service composition and execution system

which is API documentation for the Java-Eclipse Interface of Eclipse[1]. As the result, an abstract plan is stored in the FLUXPG and ready to be transformed into BPMN in prolog declaration. Then BPMN model in prolog in BPMN_PrologModel object is checked its process properties and guard condition expression in *BPMNValidation* component. We also compile and run the transform and validation programs via *com.parcltechnologies.eclipse*. If the BPMN in prolog is valid, *ParameterMonitor* and *BPMNExecution* components in Execution layer are enable. The BPMN_PrologModel object is monitored whether it possesses all process input parameters from the Requirement object or not. User is asked to submit missing parameter(s) if all the parameter inputs are not found in the Requirement object. Then BPMNModel object is created from the BPMN_PrologModel object and the Requirement object. To implement and execute BPMN model, we implement an object of *org.activiti*, which is API documentation for modeling and execution BPMN model (i.e., generating graphical BPMN process, deploying the process to the engine, starting a process instance and save process diagram to a file) [Tea]. Finally, result of BPMN model execution in BPMNModel object is sent back to the user.

[1]an open-source software system for the cost-effective development and deployment of constraint programming applications [ECL].

5.3 Results

Our main challenge to validate our framework is to find service operations of real world services that are composable in E-Tourism domain. Unfortunately, to best our knowledge, there is no service operations that allow us to fully operate the performance of our framework. Therefore, to prove the feasibility of our approach to compose service operations that fulfill user's query, we have make an implementation as a proof of concept and experiments to evaluate outputs of our automated service operations composition and execution approach. For experiments as follow, we use three motivating scenarios introduced in Chapter 1. The scenarios describe different problem situations of Alice facing during her trip in Paris. For each situation, she submits a textual query via her smart phone to the Context Awareness Recommendation system, introduced in Chapter 1, to obtain a service solution. We capture the Service Composition and Execution part of the system that is responsible for synthesizing automatically a service composition among relevant service operations, so that the execution of the composite service fulfills user's need.

Therefore, we set up each experiment having Alice's problem query in text, which its full description is found in Section 1.2, and per-processing input in propositional logics of user's profile, context, query, which is expressed as initial and goal states, and list of service operations, which each is described with its input(s) and output(s). We illustrate results of the experiments consisting of (1) a valid BPMN model in Prolog which illustrates a valid assembly of abstract service operation in BPMN specification, (2) its equivalent to graphical BPMN model, (3) missing input parameter(s) which are later filled by Alice to obtain (4) all input parameters for a service execution and (5) a result of the service execution.

5.3.1 Experiment 1.

The aim of the experiment 1 is to execute a query, which has **one operation** to fulfill a goal the query explicitly defining, on our Service Composition and Execution system. We pick up one query existing in Alice's scenario (see Section 1.2), which is *I want to search for apartment rooms available from tonight to 05/06/2014.* To set up an experiment, we prepare the following data inputs of

query, profile, context and service operations derived from the User Interaction and Query Management (UIQM) and the Discovery system (DS) respectively.

- **From UIQM**

Query	
Initials	**Goals**
CheckIn(01/06/2014) CheckOut(05/06/2014) RoomType(apartment)	RoomAvailables

Profile	**Context**
PersonName(Alice) Citizenship(USA) TravalPurpose(work)	Date(01/06/2014) Time(12.00)

- **From DS**

Service operation	**Input(s)**	**Output(s)**
SearchAvailableRooms	Location CheckIn CheckOut RoomType NumberOfGuest	RoomAvailables

These mention above pre-processing inputs are transformed into *FLUXQuery* as belows using methods in *FLUXQueryTransformer* class.

```
1  % there is one service operation in an operation list.
2  op_list(["SearchAvailableRooms"]).
3
4  % the service operation is tramsformed here
5  op_inputs("SearchAvailableRooms",["Location","CheckIn","CheckOut","RoomType",
6  "NumberOfGuest"]),
7  op_outputs("SearchAvailableRooms",["RoomAvailables"]),
8
9  % intial and goal states are defined here
10 initials(["CheckIn","CheckOut","RoomType","PersonName", "Citizenship", "TravelPurpose",
11 "Date", "Time"]),
12
13 initialsWithValues(["CheckIn:01-06-2014","CheckOut:05-06-2014","RoomType:apartment",
14 "PersonName:Alice","Citizenship:USA","TravelPurpose:work","Date:01-06-2014","Time:12am"]),
15
16 goals(["RoomAvailables"])].
```

The generated *FLUXQuery* is injected into *FLUXPlaner*. With composition rules, the *FLUXPlaner* is able to synthesizes an abstract plan of the Alice's query. The partial of Alice's abstract plan is shown below.

add_initial(SearchAvailableRooms),
add_goal(SearchAvailableRooms),
...

The above abstract plan shows that it starts and ends with SearchAvailableRooms operation. Then the Service composition and execution system transforms the abstract plan into semantic BPMN model using *BPMNTransformer* class as follows.

node(start),
node(end1),
flow(start,searchAvailableRooms),
flow(SearchAvailableRooms,end1),
task(SearchAvailableRooms,[Location, CheckIn, CheckOut, RoomType
NumberOfGuest],[RoomAvailabilities]).
parameters([CheckIn:01-06-2014, CheckOut:05-06-2014,
RoomType:apartment]),
missingP(NumberOfGuest), *missingP*(Location)

The generated semantic business model consists of three elements (one start event node(start), one task operation SearchAvailableRooms and one end event node(end1)) and the flows connecting those elements. Besides, we keep the process input parameters of CheckIn, CheckOut, RoomType and missing parameters of NumberOfGuest, Location in the model.

Next Alice is asked to submit the mentioned missing parameter(s). Alice puts $48.8567°N2.3508°E$ for Location and *2* for NumberOfguest. Thus all required parameters with values are presented here.

<InputParameters>
<Location> $48.8567°N2.3508°E$) < /Location>
<CheckIn> $01/06/2014$ < /CheckIn>
<CheckOut> $05/06/2014$ < /CheckOut>
<NumberOfguest> 2 < /NumberOfguest>
<InputParameters>

After the BPMN model is implemented into BPMN format and it is deployed on the server. We then execute the model with the InputParameters. The BPMN diagram of Alice's query is shown below.

FIGURE 5.8: SearchAvailableRooms_process

Finally, the resulting of BPMN model execution holding three set of RoomAvailable information as follows is sent back to Alice.

<RoomAvailables>
<RoomAvailable>
<Id>*Sofa bed in the heart of paris*< /Id>
<Type>*Shared room*< /Type>
<Address>*Rue des Rosiers, Paris 4e arrondissement, Ile-de-France 75004*< /Address>
<City>*Paris*< /City>
<Price>*EUR38*< /Price>
<CheckIn>*01/06/2014*< /CheckIn>
<CheckOut>*05/06/2014*< /CheckOut>
<NumberOfGuest>*2*< /NumberOfGuest>
< /RoomAvailable>
<RoomAvailable>
<Id>*Nice flat in Montmarte*< /Id>
<Type>*Private room*< /Type>
<Address>*Rue Lamarck, Paris, Ile-de-France 75018*
< /Address>
<City>*Paris*< /City>
<Price>*EUR47*< /Price>
<CheckIn>*01/06/2014*< /CheckIn>
<CheckOut>*05/06/2014*< /CheckOut>
<NumberOfGuest>*2*< /NumberOfGuest>

< /RoomAvailable>
<RoomAvailable>
<Id>*Quiet, Spacious apt in City Center*< /Id>
<Type>*Entire home*< /Type>
<Address>*Rue Dussoubs, Paris, Ile-de-France 75002*
< /Address>
<City>*Paris*< /City>
<Price>*EUR132*< /Price>
<CheckIn>*01/06/2014*< /CheckIn>
<CheckOut>*05/06/2014*< /CheckOut>
<NumberOfGuest>*2*< /NumberOfGuest>
< /RoomAvailable>
< /RoomAvailables>

5.3.2 Experiment 2.

The aim of the experiment 2 is to execute a query, which has sequencing and paralleling among operations, so that the composite operation fulfills the defined goals, on our Service Composition and Execution system. We pick up one query existing in Alice's scenario (see Section 1.2), which is *I want to book a table for 2 people at the finest restaurant at 8pm. in the city, and the direction to the restaurant.* To set up an experiment, we prepare the following data inputs of query, profile, context and service operations derived from the User Interaction and Query Management (UIQM) and the Discovery system (DS) respectively.

- **From UIQM**

Query	
Initials	**Goals**
NumberOfGuest(2) RestaurantType(finest) BookingTime(8pm) BookingDate(01/06/2014)	BookingTableReservation Direction

Profile	**Context**
PersonName(Alice) Citizenship(USA) TravalPurpose(work)	Date(01/06/2014) Time(16.00)

- **From DS**

Operation Name	**Input(s)**	**Output(s)**
FromCoorinateToCity	Coordinates	City
FindFinestRestaurant	City	RestaurantID RestaurantName RestaurantAddress
BookRestaurant	RestaurantName BookingTime BookingDate NumberOfGuest GuestName GuestTelephone	BookingTableReservation
GetDirection	FromAddress ToAddress	Direction

These mention above pre-processing inputs are transformed into *FLUXQuery* as belows using methods in *FLUXQueryTransformer* class.

```
% there are four service operations in an operation list.
op_list(["FromCoorinateToCity","FindFinestRestaurant","BookRestaurant","GetDirection"]).

% the service operation is tramsformed here.
op_inputs("FromCoorinateToCity",["Coordinates"]),
op_outputs("FromCoorinateToCity",["City"]),
op_inputs("FindFinestRestaurant",["City"]),
op_outputs("FindFinestRestaurant",["RestaurantID","RestaurantName","RestaurantAddress"]),
op_inputs("BookRestaurant",["RestaurantName","BookingTime","BookingDate","NumberOfGuest",
"GuestName","GuestTelephone"]),
op_outputs("BookRestaurant",["BookingTableReservation"]),
```

```
12 op_inputs("GetDirection",["FromAddress","ToAddress"]),
13 op_outputs("GetDirection",["Direction"]),
14
15 % list of data transform is presented here.
16 dataTransform(RestaurantAddress,ToAddress),
17 dataTransform(Coordinates,ToAddress),
18
19 % intial and goal states are defined here.
20 initials(["NumberOfGuest","RestaurantType","BookingTime","BookingDate","Citizenship",
21 "TravelPurpose", "Date", "Time"]),
22
23 initialsWithValues(["NumberOfGuest:2","RestaurantType:finest","BookingTime:8pm",
24 "BookingDate:01-06-2014","PersonName:Alice","Citizenship:USA","TravelPurpose:work",
25 "Date:01-06-2014","Time:12am"]),
26
27 goals(["BookingTableReservation","Direction"])].
```

The generated *FLUXQuery* is injected into *FLUXPlaner*. With composition rules, the *FLUXPlaner* is able to synthesizes an abstract plan of the Alice's query as follows.

add_initial(FromCoorinateToCity),
flow(FromCoorinateToCity,FindFinestRestaurant),
flow(FindFinestRestaurant,BookRestaurant),
flow(FindFinestRestaurant,GetDirection),
add_goal(BookRestaurant),
add_goal(GetDirection),
...

The above abstract plan shows that it starts with FromCoorinateToCity operation and ends with two operations BookRestaurant and GetDirection. The order of its workflow begin with FromCoorinateToCity following by FindFinestRestaurant operation, which connects to two operations BookRestaurant and GetDirection at the same time. Then the Service composition and execution system transforms the abstract plan into semantic BPMN model using *BPMNTransformer* class as follows.

node(start),
gateway(andS1),
node(end1), *node*(end2),
flow(start,FromCoorinateToCity),

flow(FromCoorinateToCity,FindFinestRestaurant),
flow(FindFinestRestaurant,andS1),
flow(andS1,BookRestaurant),
flow(andS1,GetDirection),
flow(BookRestaurant,end1),
flow(GetDirection,end2),
task(FromCoorinateToCity,[Coordinates],[City]),
task(FindFinestRestaurant,[City],
[RestaurantID,RestaurantName,RestaurantAddress]),
task(BookRestaurant,[RestaurantName,BookingTime,BookingDate,
NumberOfGuest,GuestName,GuestTelephone],
[BookingTableReservation]),
task(GetDirection,[Coordinates,Address],[Direction]),
dataTransform(RestaurantAddress,ToAddress),
dataTransform(Coordinates,ToAddress).
parameters([BookingTime:8pm, BookingDate:01-06-2014,
NumberOfGuest:2]),
missingP(Coordinates), *missingP*(GuestName),
missingP(GuestTelephone)

The generated semantic business model consists of eight elements (one start event node(start), four task operations FromCoorinateToCity, FindFinestRestaurant, BookRestaurant and GetDirection, one parallel gateway gateway(andS1), two end events node(end1)) and node(end2)) and the flows connecting those elements. Besides, we keep the process input parameters of BookingTime, BookingDate, NumberOfGuest and missing parameters of Coordinates, GuestName, GuestTelephone in the model.

Next Alice is asked to submit the mentioned missing parameter(s). Alice puts $48.8567°N2.3508°E$ for Coordinates, *Alice* for GuestName and *078956120* for GuestTelephone. Thus all required parameters with values are presented here.

<InputParameters>
<Coordinates> $48.8567°N2.3508°E$) < /Coordinates>
<BookingTime> $20:00$ < /BookingTime>
<BookingDate> $05/06/2014$ < /BookingDate>
<NumberOfguest> 2 < /NumberOfguest>

<GuestName> *Alice* < /GuestName>
<GuestTelephone> 078956120 < /GuestTelephone>
<InputParameters>

After the BPMN model is implemented into BPMN format and it is deployed on the server. We then execute the model with the InputParameters. The below BPMN diagram of Alice's query shows sequencing and paralleling among operations.

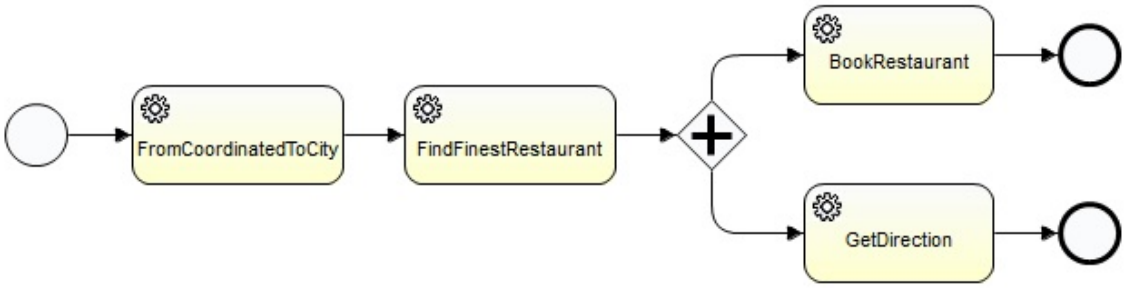


FIGURE 5.9: FindRestaurantAndDirection_process

Finally, the resulting of BPMN model execution holding BookingTableReservation and Direction information as follows is sent back to Alice.

<BookingTableReservation>
<Status>*booked*< /Status>
<RestaurantResult>
<RestaurantName>*Chez Marie Louise*
< /RestaurantName>
<BookingTime>*20:00*< /BookingTime>
<BookingDate>*01/03/2014*< /BookingDate>
<NumberOfguest>*2*< /NumberOfguest>
<GuestName>*Alice*< /GuestName>
<GuestTelephone>*078956120*< /GuestTelephone>
<NumberOfGuest>*2*< /NumberOfGuest>
< /RestaurantResult>
< /BookingTableReservation>
<Direction>
48.850266, 2.330650, Head north on Rue Cassette toward Rue de Mézières, Take Boulevard Saint-Germain, Boulevard de Sébastopol

and Boulevard de Strasbourg to Rue des Récollets, Continue to Rue Bichat, Continue on Rue Bichat. Drive to Rue Marie et Louise, Chez Marie Louise 11 Rue Marie et Louise, 75010 Paris
< /Direction>

5.3.3 Experiment 3.

The aim of the experiment 3 is to execute a query, which has conditioning among operations, so that the composite operation fulfills the defined goals, on our Service Composition and Execution system. We pick up one query existing in Alice's scenario (see Section 1.2), which is *I want to buy a ticket for Euro Disney tomorrow if the weather forecast is sunny, otherwise, buy a ticket for Louvre museum.* To set up an experiment, we prepare the following data inputs of query, profile, context and service operations derived from the User Interaction and Query Management (UIQM) and the Discovery system (DS) respectively.

- **From UIQM**

Query	
Initials	**Goals**
ForecastDate(02/06/2014)	TicketLouveMuseum Condition(if(Weather /== Sunny)) TicketEuroDisney Condition(if(Weather == Sunny)

User's Profile	**User's Context**
PersonName(Alice)	Coordinates(48.8567° N 2.3508° E)
Citizenship(USA)	Date(01/06/2014)
TravalPurpose(work)	Time(12.00)

- **From DS**

Operation Name	**Input(s)**	**Output(s)**
GetWeatherForecast	Coordinates ForecastDate	Weather
BuyTicketLouveMuseum	NumberOfTicket TicketDate TicektTime CreditcardInfo	TicketLouveMuseum
BuyTicketEuroDisney	NumberOfTicket TicketDate TicektTime CreditcardInfo	TicketEuroDisney

These mention above pre-processing inputs are transformed into *FLUXQuery* as belows using methods in *FLUXQueryTransformer* class.

```
% there are three service operations in an operation list.
op_list(["GetWeatherForecast","BuyTicketLouveMuseum","BuyTicketEuroDisney"]).

% the service operation is tramsformed here.
op_inputs("GetWeatherForecast",["Coordinates","ForecastDate"]),
op_outputs("GetWeatherForecast",["Weather"]),
op_inputs("BuyTicketLouveMuseum",["NumberOfTicket","TicketDate","TicektTime",
"CreditcardInfo"]),
op_outputs("BuyTicketLouveMuseum",["TicketLouveMuseum"]),
op_inputs("BuyTicketEuroDisney",["NumberOfTicket","TicketDate","TicektTime",
"CreditcardInfo"]),
op_outputs("BuyTicketEuroDisney",["TicketEuroDisney"]),

% intial and goal states are defined here.
initials(["NumberOfGuest","RestaurantType","BookingTime","BookingDate","Citizenship",
"TravelPurpose", "Date", "Time"]),

initialsWithValues(["NumberOfGuest:2","RestaurantType:finest","BookingTime:8pm",
"BookingDate:01-06-2014","PersonName:Alice","Citizenship:USA","TravelPurpose:work",
"Date:01-06-2014","Time:12am"]),

goals(["TicketEuroDisney","TicketLouveMuseum"])].
cond(c1,if("Weather","sunny","TicketEuroDisney"),
else("Weather","rainny",["BookingRestaurantReservation","Direction"]))
```

The generated *FLUXQuery* is injected into *FLUXPlaner*. With composition rules, the *FLUXPlaner* is able to synthesizes an abstract plan of the Alice's query as follows.

add_initial(GetWeatherForecast),

flow(GetWeatherForecast,*if*(weather,"==","sunny",BuyTicketEuroDisney),
else(weather,"!=","sunny",BuyTicketLouveMuseum)),
add_goal(BuyTicketEuroDisney),
add_goal(BuyTicketLouveMuseum),
...

The above abstract plan shows that it starts with GetWeatherForecast operation and ends with two operations BuyTicketEuroDisney and BuyTicketLouveMuseum. There is one condition flow in the plan specifying that if the weather is predicted as sunny, the BuyTicketEuroDisney operation is enable or else the BuyTicketLouveMuseum operation is enable. Then the Service composition and execution system transforms the abstract plan into semantic BPMN model using *BPMNTransformer* class as follows.

node(start),
gateway(xorS1),
node(end1), *node*(end2),
flow(start,GetWeatherForecast),
flow(GetWeatherForecast,xorS1),
flow(xorS1,BuyTicketLouveMuseum,*if*(Weather,==,sunny)),
flow(xorS1,BuyTicketEuroDisney),*if*(Weather,/==,sunny)
flow(BuyTicketLouveMuseum,end1),
flow(BuyTicketEuroDisney,end2),
task(GetWeatherForecast,[Coordinates,ForecastDate],[Weather]),
task(BuyTicketLouveMuseum,[NumberOfTicket,TicketDate,
CreditcardInfo],[TicketLouveMuseum]),
task(BuyTicketEuroDisney,[NumberOfTicket,TicketDate,
CreditcardInfo],[TicketEuroDisney]),
parameters([ForecastDate:02-26-2014, NumberOfTicket:2,
TicketDate:02-26-2014]),
missingP(Coordinates), *missingP*(CreditcardInfo),

The generated semantic business model consists of seven elements (one start event node(start), three task operations GetWeatherForecast, BuyTicketLouveMuseum, BuyTicketEuroDisney, one condition gateway gateway(xorS1), two end events node(end1)) and node(end2)) and the flows connecting those elements. Besides, we

keep the process input parameters of ForecastDate, NumberOfTicket, TicketDate and missing parameters of Coordinates, CreditcardInfo in the model.

Next Alice is asked to submit the mentioned missing parameter(s). Alice puts $48.8567°N2.3508°E$ for Coordinates and *Visa no. 548910578XXX ...* for CreditcardInfo. Thus all required parameters with values are presented here.

<InputParameters>
<Coordinates> $48.8567°N2.3508°E$) < /Coordinates>
<ForecastDate> 02/06/2014 < /ForecastDate>
<NumberOfTicket> 2 < /NumberOfTicket>
<TicketDate> 02/06/2014 < /TicketDate>
<CreditcardInfo>*Visa no. 548910578XXX ...*< /CreditcardInfo>
<InputParameters>

After the BPMN model is implemented into BPMN format and it is deployed on the server. We then execute the model with the InputParameters. The below BPMN diagram of Alice's query shows conditioning among operations.

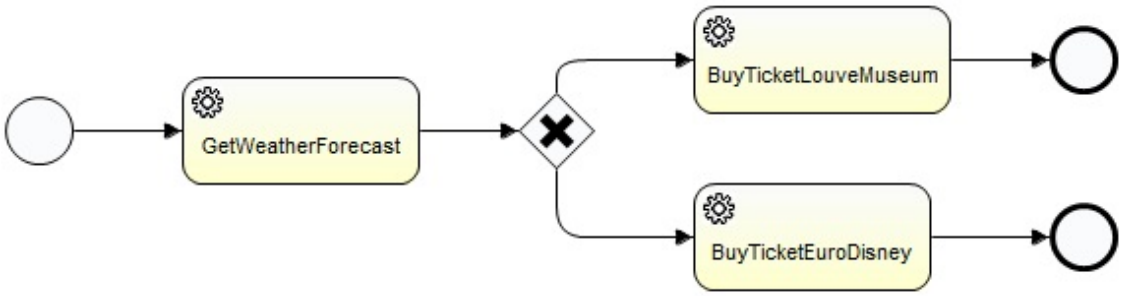


FIGURE 5.10: GetWeatherForecastAndBuyTicket_process

There are two alternative solutions of the resulting of BPMN model execution. One is for condition that *If Weather returns "NOT Sunny"*:

<TicketLouveMuseum>
<Status>*paid*< /Status>
<Tickets>
<Ticket>
<TicketID>*0121*< /TicketID>
<TicketType>*single*< /TicketType>

```
            <TicketDate>02/06/2014< /TicketDate>
        < /Ticket>
        <Ticket>
            <TicketID>0122< /TicketID>
            <TicketType>single< /TicketType>
            <TicketDate>02/06/2014< /TicketDate>
        < /Ticket>
    < /Tickets>
< /TicketLouveMuseum>
```

The other is for condition that *If Weather returns "Sunny"*:

```
<TicketEuroDisney>
    <Status>paid< /Status>
    <Tickets>
        <Ticket>
            <TicketID>0514< /TicketID>
            <TicketType>family< /TicketType>
            <TicketDate>02/06/2014< /TicketDate>
        < /Ticket>
    < /Tickets>
< /TicketEuroDisney>
```

5.4 Discussion

We performed the evaluation for the correctness of our Automated Service Composition and Execution approach on three running scenarios from Chapter 1. For Experiment 1, a result is shown a finding of a single service operation SearchAvailableRooms and execution of a BPMN process invoking the discovered service operation in Figure 5.8. A result of service execution in xml document contains root element as RoomAvailables which matches to predefined RoomAvailables in Alice's goal. For Experiment 2, a result in Figure 5.9 shows a composite model supporting sequencing and paralleling constructs among service operations: FromCoordinatesToCity, FindFinestRestaurant, BookRestaurant and GetDirection operations. A result

of service execution in xml document contains root element as BookingRestaurantResveration,Direction which matches to predefined BookingRestaurantResveration,Direction in Alice's goals. For lastly Experiment 3, a result in Figure 5.10 shows a composite model supporting conditioning constructs among service operations: GetWeatherForecast, BuyTicketLouveMuseum and BuyTicketEuroDisney operations. A result of this service execution has two optional outputs. It returns a result in xml document containing root element as TicketLouveMuseum if the GetWeatherForecast operation return no Sunny for Weather value, which matches to predefined TicketLouveMuseum, Condition(if(Weather /== Sunny)) in Alice's goals. Whereas another result in xml document containing root element as TicketEuroDisney is returned if the GetWeatherForecast operation return Sunny for Weather value, which satisfies with Alice's query expression TicketEuroDisney, Condition(if(Weather == Sunny)).

Chapter 6

Conclusion and Future work

Contents

Abstract. In this Chapter, we conclude the research and offers future research recommendations.

In this dissertation, we investigated a problem of Automated Service Composition and Execution in static execution environment. We proposed a framework for (1) composing automatic service operations, so that a BPMN composition model fulfills both BPMN specification and defined user's goals at the same time and (2) executing the BPMN model with user's parameters, so that the result of service execution returns to the user.

On the basis of our composition technique, we defined user requirements expressed in a sequence of goal states in propositional logic. We show how our planning algorithm is working with abstract goal states to search and connect to service operations, by mean of service operation annotations, for a solution. The solution is expected for workflow model of sequencing, paralleling and conditioning among service operations. To allow for more expressive control flow requirements, we proposed our own abstract (context based) language that introduces the following features: the ability to express both reachability and procedural goals and the ability to set preferences among alternative goals. On the basis of our BPMN transformation technique, we defined reasonable semantics for BPMN elements

for mapping context-based composition model into BPMN model in Prolog declaration. We proposed a new planning algorithm and exploited planning-as-model-checking approach developing the idea of [MPT08] to validate the Prolog BPMN model whether it is Well-formed and Well-defined BPMN process.

On the basis of our service execution technique, we implemented algorithm to checking missing parameters from the derived initial states and we implemented BPMN diagram and BPMN model in xml format from open source BPMN Activiti API. Finally, we executed the BPMN model with the input parameters and returned the result of the execution to the user.

One of the key contributions of the thesis is the Service Composition and Execution demonstration platform. In it, we modeled a pervasive system based on the E-Tourism scenarios and realized workflow composition, BPMN transformation, validation and implementation and execution of BPMN models using our Service Composition and Execution framework.

The platform allowed us to evaluate not only the integrability of multiple domains for our Service Composition and Execution approach but also the performance of the composition and execution engines on dynamic related scenarios.

Through out this research, we found the added values and limitations of our work. The proposed service composition and execution system framework is a generic approach suitable for most domains on component integration. The prototype developed in this thesis is for the module Composition and Orchestration System (COS) which will be integrated in the E-Tourism system. Additionally, the prototype is designed to handle both push and pull modes in the E-Tourism system. However, our framework relies only on function call (input(s) and output(s)) of service operation annotations. We did not use other service informations such as preconditions and effects, which allow users to make more complex queries. Furthermore, our composition approach supports only automatic tasks such as service tasks and script tasks in business process model. Besides, the business process model does not handle its non-functional features (i.e., artifacts and association) and organizational modeling features (i.e., lanes and pools). Lastly, we do not support process monitoring and controlling when our process engine executes the business process model.

As a matter of fact, within this dissertation we could not handle every single issue related to the topics of interest. We admit that there are still many extensions

and improvements are needed to make the approaches more mature and effective. The following we shall discuss the most important issues we plan to take in the near future.

6.1 Future work

System Assessment

There are two possibilities to assess our system. The first possibility is to run the experiment to see how many failure time of resulting composite service execution. The second possibility is to evaluate the project system from user satisfaction. To do this, we integrate our service composition and execution system into the E-tourism project.

User Interface Composition

As stated above in our limitation that our approach handles only service tasks and script task. In the future, we plan to and extend semantic user task and trigger event on our context process model to dynamically generate user interface in the business process. This user interface can keep the user in a loop until the predefined trigger of some events occur.

Complete the process

We plan to work more on user query for supporting loop control construct by encoding loop control definition into our planning domain. Furthermore, we plan to support possible exceptions happened in the process model. To do that, we might refer to experts on particular domains to add rules for the possible exceptions.

Statefull Web services

One of the direction for future work is the further development of the statefull web services. Since some users require a retainment of particular objects during process execution. Shopping cart object in Shopping online application can be one of examples that the users need to keep certain states of such objects during browsing more products. We plan to extend semantic transaction model along with our context process model.

Data Flow Requirements

We plan to continue our work on data-flow requirements, which are not completely integrated to the approach. We also consider a problem of miss-matched data

type between two service operations. For example, a pre-defined context output of one operation may return value in Interger data type while the same pre-defined context input of continuous service operation requires value in String data type.

Adaptation Process

There are a few directions for improving our process adaptation approach. One of them is proactive service operation substitution. In this work a list of service operations are delivered by another work from the Service Discovery system, which performs the most intuitive way of strategy selection. The service operation substitution can happen when we detect a failure of a service operation invocation, we request to the Service Discovery system for candidates of such service operation.

Appendix A

Planning model for the service composition agent

A.1 A list of fluents for conducting the Flux query

Initial(X):

PARAMETER $\mapsto$ FLUENT

Goal(Xs):

SET OF PARAMETER $\mapsto$ FLUENT

Op_inputs(X,Ys):

OPERATION $\times$ SET OF PARAMETER $\mapsto$ FLUENT

Op_outputs(X,Ys):

OPERATION $\times$ SET OF PARAMETER $\mapsto$ FLUENT

Cond(C, if(O1,O3), else(O2,O3)):

INDEX $\times$ PARAMETER $\times$ PARAMETER $\times$
PARAMETER $\times$ PARAMETER $\mapsto$ FLUENT

Cond(C, if(O1,O3), else(O2,O3)):

INDEX $\times$ PARAMETER $\times$ PARAMETER $\times$
PARAMETER $\times$ PARAMETER $\mapsto$ FLUENT

DataTransform(RestaurantAddress,ToAddress):

PARAMETER $\times$ PARAMETER $\mapsto$ FLUENT

A.2 A list of fluents for the abstract plan

Add_initail(X):

OPERATION $\mapsto$ FLUENT

Flow(X,Y):

OPERATION $\times$ OPERATION $\mapsto$ FLUENT

Add_goal(X):

OPERATION $\mapsto$ FLUENT

Flow(X, *if*(*guard*(O1,RO1,V1),Y1),*else*(*guard*(O2,RO2,V2),Y2)):

OPERATION $\times$ PARAMETER $\times$ OPERAND $\times$ VALUE $\times$
OPERATER $\times$ PARAMETER $\times$ OPERAND $\times$ VALUE $\times$
OPERATER $\mapsto$ FLUENT

Flow(*if*[X1,X2],Y):

OPERATION $\times$ OPERATION $\times$ OPERATION $\mapsto$ FLUENT

A.3 A list of actions the agent performing the abstract plan

Add_initial(X,Z1,Z2):
VALUE $\mapsto$ ACTION

Add_flow(Z1,X,Y,Z2):
VALUE $\times$ VALUE $\mapsto$ ACTION

Add_goal(Z,X,Z2) :
VALUE $\mapsto$ ACTION

Add_OR_split(Z1,X,O1,RO1,V1,Y1,O2,RO2,V2,Y2,Z2):
VALUE $\times$ VALUE $\times$ VALUE $\times$ VALUE $\times$ VALUE $\times$ VALUE $\times$
VALUE $\times$ VALUE $\mapsto$ ACTION

Add_OR_join(Z1,X1,X2,Y,Z2)
VALUE × VALUE × VALUE ↦ ACTION

Appendix B

Program for the service composition agent

```
:- ['6_flux'].
:- lib(xml).

:- discontiguous(state_update/4).

add_flow(Z,Opx,Opy,Z1) :-
        state_update(Z,add_flow(Opx,Opy),Z1,[]).

add_goal(Z,Op,Z2)       :-
        state_update(Z,add_goal(Op),Z2,[]).

add_initial(Op,Z,Zn)    :-
        state_update(Z,add_initial(Op),Zn,[]).

add_OR_split(Z,Op1,O1,S1,V1,Op2,O2,S2,V2,Op3,Z2)        :-
        state_update(Z,add_or_split(Op1,O1,S1,V1,Op2,O2,S2,V2,Op3),Z2,[]).

add_OR_join(Z,Op1,Op2,Op3,Z2)   :-
        state_update(Z,add_or_join(Op1,Op2,Op3),Z2,[]).



state_update(Z,add_flow(Opx,Opy),Z1,[]) :-
        update(Z,[flow(Opx,Opy)],[],Z1).

state_update(Z,add_goal(Op),Z2,[]) :-
        update(Z,[add_goal(Op)],[],Z2).

state_update(Z,add_initial(Op),Zn,[]) :-
        update(Z,[add_initial(Op)],[],Zn).

state_update(Z,add_or_split(Op1,O1,S1,V1,Op2,O2,S2,V2,Op3),Zn,[]) :-
        not_holds_all(flow(Op1,_),Z) ->
        update(Z,[flow(Op1,if(O1,S1,V1,Op2),else(O2,S2,V2,Op3)),add_initial(Op1)],
```

```
35          [flow(Op1,Op2),flow(Op1,Op3),add_initial(Op2),add_initial(Op3)],Zn)
36                                                                      ;
37          update(Z,[flow(Op1,if(O1,S1,V1,Op2),else(O2,S2,V2,Op3))],
38          [flow(Op1,Op2),flow(Op1,Op3),add_initial(Op2),add_initial(Op3)],Zn).
39
40  state_update(Z,add_or_join(Op1,Op2,Op3),Z2,[]) :-
41          update(Z,[flow(if[Op1,Op2],Op3)],[add_initial(Op3)],Z2).
42
43
44  init(Z0) :-
45
46  Z0 = [op_inputs(fCC,['coordinates']),op_inputs(fFR,['city']),
47          op_inputs(gD,['addressFrom','addressTo']),
48          op_outputs(fCC,['city']),
49          op_outputs(fFR,['name(restaurant)','address(restaurant)']),
50          op_outputs(bR,['bookingConfirm']),
51          op_outputs(gD,['direction']),
52          dataTransform('coordinates','addressFrom'),
53          dataTransform('address(restaurant)','addressTo'),
54          initials([coordinates, number(guest),date]),
55          goals(['name(restaurant)','direction'])].
56
57  op_list([fCC,fFR,gD]).
58  condition_list([]).
59
60  main3 :-
61  init(Z0),op_list(Op_list1),op_list(Op_list2),knows_val([Gs],goals(Gs),Z0),
62  condition_list(CList), knows_val([Initials],initials(Initials),Z0),
63  (check_flows(Op_list1,Op_list2,Z0,Z1) -> Fflow=true;Fflow=false, Z0=Z1),
64  check_goals(Gs,Op_list1,Z1,Z2), check_initial(Op_list1,Z2,Z3),
65  check_OR_split2(CList,Op_list1,Op_list2,Z3,Z4),check_OR_join(CList,Z4,Z5),
66  create_initial(Op_list1,N,P1,Z5,Z6),create_ends(Z6,Op_list1,N,P1,P2,Z7),
67  create_gateways(Op_list1,Op_list2,N,L1,L2,P2,P3,Z7,Z8),writeln(Z8).
68
69
70  check_flows([],_,Z,Z).
71  check_flows([Opx|Op_list1],Op_list2,Z1,Z)       :-
72          knows_val([Outputs],op_outputs(Opx,Outputs),Z1),
73          check_outputs(Outputs,Opx,Op_list2,Z1,Z2),
74          check_flows(Op_list1,Op_list2,Z2,Z).
75
76  check_outputs([],_,_,Z,Z).
77  check_outputs([X|Xs],Opx,Op_list,Z1,Zn) :-
78          check(X,Opx,Op_list,Z1,Z2),
79          check_outputs(Xs,Opx,Op_list,Z2,Zn).
80
81  check(_,_,[],Z,Z).
82  check(Output,Opx,[Opy|Op_list],Z,Zn)    :-
83          knows_val([Inputs],op_inputs(Opy,Inputs),Z),del([],Inputs,Input1),
84          ((member(Output,Input1) -> (add_flow(Z,Opx,Opy,Z1),
85                                     check(Output,Opx,Op_list,Z1,Zn)))
86                                     ;
87          (knows_val([Output1],dataTransform(Output,Output1),Z),
88           member(Output1,Input1) -> (add_flow(Z,Opx,Opy,Z1),
89                                 check(Output,Opx,Op_list,Z1,Zn)))
```

```
90                                  ;check(Output,Opx,Op_list,Z,Zn)).
91
92
93  check_goals(_,[],Z,Z).
94  check_goals(Gs,[Op|Op_list1],Z,Zn)      :-
95          knows_val([Outputs],op_outputs(Op,Outputs),Z), check_goal(Gs,Outputs,Op,Z,Z1),
96          check_goals(Gs,Op_list1,Z1,Zn).
97
98
99  check_goal([],_,_,Z,Z).
100 check_goal([G|Gs],Outputs,Op,Z,Zn) :-
101         ((member(G,Outputs),holds(flow(_,Op),Z), not_holds_all(flow(Op,_),Z))
102                                 -> (add_goal(Z,Op,Z1), check_goal(Gs,Outputs,Op,Z1,Zn))
103                                 ;
104         (member(G,Outputs),not_holds_all(flow(_,Op),Z),not_holds_all(flow(Op,_),Z))
105                                 -> (add_goal(Z,Op,Z1), add_initial(Op,Z1,Z2),
106                                 check_goal(Gs,Outputs,Op,Z2,Zn))
107                                 ; check_goal(Gs,Outputs,Op,Z,Zn)).
108
109
110 check_initial([],Z,Z).
111 check_initial([Op|Op_list1],Z,Zn) :-
112         ((holds(flow(Op,_),Z), not_holds_all(flow(_,Op),Z)))
113                         -> (add_initial(Op,Z,Z1), check_initial(Op_list1,Z1,Zn))
114                         ;check_initial(Op_list1,Z,Zn).
115
116
117 check_OR_split([],Z,Z).
118 check_OR_split([C|Cs],Z,Zn)     :-
119         knows_val([O1,S1,V1,T1,O2,S2,V2,T2],cond(C,if(O1,S1,V1,T1),else(O2,S2,V2,T2)),Z)
120                                         ->
121                 (knows_val([Op1],op_outputs(Op1,[O1|_]),Z),knows_val([Op2],
122                 op_outputs(Op2,[T1|_]),Z),knows_val([Op3],op_outputs(Op3,[T2|_]),Z),
123                 add_OR_split(Z,Op1,O1,S1,V1,Op2,O2,S2,V2,Op3,Z2),
124                 check_OR_split(Cs,Z2,Zn))
125                                         ; check_OR_split(Cs,Z,Zn).
126
127 check_OR_split1([],_,_,Z,Z).
128 check_OR_split1([C|Cs],Op_list1,Op_list2,Z,Zn)  :-
129         knows_val([O1,S1,V1,T1,O2,S2,V2,T2],cond(C,if(O1,S1,V1,T1),else(O2,S2,V2,T2)),Z)
130                                         ->
131                 (knows_val([Op1],op_outputs(Op1,[O1|_]),Z),
132                 knows_val([Op2],op_outputs(Op2,[T1|_]),Z),
133                 check_Head1(Op_list1,Op_list2,Op2,Op2h,Z),
134                 knows_val([Op3],op_outputs(Op3,[T2|_]),Z),
135                 check_Head1(Op_list1,Op_list2,Op3,Op3h,Z),
136                 add_OR_split(Z,Op1,O1,S1,V1,Op2h,O2,S2,V2,Op3h,Z2),
137                 check_OR_split1(Cs,Op_list1,Op_list2,Z2,Zn))
138                                         ; check_OR_split1(Cs,Op_list1,Op_list2,Z,Zn).
139
140 check_OR_split2([],_,_,Z,Z).
141 check_OR_split2([C|Cs],Op_list1,Op_list2,Z,Zn)  :-
142         knows_val([O1,V1,T1,O2,V2,T2],cond(C,if(O1,V1,T1),else(O2,V2,T2)),Z),
143         knows_val([Op1],op_outputs(Op1,[O1|_]),Z),
144         (((not(isList(T1)),not(isList(T2))) -> (knows_val([Op2],op_outputs(Op2,[T1|_]),Z),
```

```
145                 check_Head1(Op_list1,Op_list2,Op2,Op2h,Z),
146                 knows_val([Op3],op_outputs(Op3,[T2|_]),Z),
147                 check_Head1(Op_list1,Op_list2,Op3,Op3h,Z),
148                 add_OR_split(Z,Op1,O1,S1,V1,Op2h,O2,S2,V2,Op3h,Z2),
149                 check_OR_split2(Cs,Op_list1,Op_list2,Z2,Zn)))
150                                                         ;
151         ((isList(T1),not(isList(T2))) -> (knows_val([Op3],op_outputs(Op3,[T2|_]),Z),
152                 check_Head1(Op_list1,Op_list2,Op3,Op3h,Z),
153                 check_Head3(T1,Op_list1,Op_list2,Th,H,Z),
154                 del([],H,Hn),
155                 (is_set(Hn) -> add_OR_split(Z,Op1,O1,S1,V1,T1,O2,S2,V2,Op3h,Z2)
156                                         ;
157                                         last(Hn,H1),
158                                         add_OR_split(Z,Op1,O1,S1,V1,H1,O2,S2,V2,Op3h,Z2),
159                                         check_OR_split2(Cs,Op_list1,Op_list2,Z2,Zn))))
160                                                 ;
161         ((not(isList(T1)),isList(T2)) -> (knows_val([Op2],op_outputs(Op2,[T1|_]),Z),
162                 check_Head1(Op_list1,Op_list2,Op2,Op2h,Z),
163                 check_Head3(T2,Op_list1,Op_list2,Th,H,Z),
164                 del([],H,Hn),
165                 (is_set(Hn) -> add_OR_split(Z,Op1,O1,S1,V1,Op2h,O2,S2,V2,T2,Z2)
166                                         ;
167                                         last(Hn,H1),
168                                         add_OR_split(Z,Op1,O1,S1,V1,Op2h,O2,S2,V2,H1,Z2),
169                                         check_OR_split2(Cs,Op_list1,Op_list2,Z2,Zn))))
170                                                                 ;
171         ((isList(T1),isList(T2))-> (check_Head3(T1,Op_list1,Op_list2,H,H1,Z),
172                 check_Head3(T2,Op_list1,Op_list2,H,H2,Z),
173                 del([],H1,H1n),del([],H2,H2n), ((is_set(H1n),
174                 is_set(H2n)) -> add_OR_split(Z,Op1,O1,S1,V1,T1,O2,S2,V2,T2,Z2)
175                                         ;
176                                         last(H1n,H11),last(H2n,H21),
177                                         add_OR_split(Z,Op1,O1,S1,V1,H11,O2,S2,V2,H21,Z2),
178                                         check_OR_split2(Cs,Op_list1,Op_list2,Z2,Zn)))))).
179
180
181 check_Head1([],_,Op2,Op2,Z).
182 check_Head1([Op|Op_list1],Op_list2,Op2,Opn,Z)   :-
183         knows_val([Op],flow(Op,Op2),Z)
184                                         ->  check_Head2(Op_list1,Op_list2,Op,Opn,Z)
185                                         ;  check_Head1(Op_list1,Op_list2,Op2,Opn,Z).
186
187 check_Head2(Op_list1,[],Op2,Op2,Z).
188 check_Head2(Op_list1,[Opx|Op_list2],Op2,Opn,Z)  :-
189         knows_val([Opx],flow(Opx,Op2),Z)
190                                         ->  check_Head2(Op_list1,Op_list2,Opx,Opn,Z)
191                                         ;  check_Head2(Op_list1,Op_list2,Op2,Opn,Z).
192
193 check_Head3([],Op_list1,Op_list2,H,H,Z).
194 check_Head3([T|T2],Op_list1,Op_list2,H,H2,Z) :-
195         knows_val([Op2],op_outputs(Op2,[T|_]),Z),
196         check_Head1(Op_list1,Op_list2,Op2,Op2h,Z), H1 = [Op2h|H],
197         check_Head3(T2,Op_list1,Op_list2,H1,H2,Z).
198
199 check_OR_join([],Z,Z).
```

```
200 check_OR_join([C|Cs],Z,Zn)      :-
201         knows_val([O1,O2,T1],cond(C,if(O1,T1),else(O2,T1)),Z)
202                         ->
203                 (knows_val([Op1],op_outputs(Op1,[O1|_]),Z), knows_val([Op2],
204                 op_outputs(Op2,[O2|_]),Z),knows_val([Op3],op_outputs(Op3,[T1|_]),Z),
205                 add_OR_join(Z,Op1,Op2,Op3,Z2),
206                 check_OR_join(Cs,Z2,Zn))
207                         ;
208                 check_OR_join(Cs,Z,Zn).
209
210 last([Elem], Elem).
211 last([_|Tail], Elem) :- last(Tail, Elem).
212
213 isList([]).
214 isList([_|Tail]):-
215         isList(Tail).
216
217 is_set(Lst) :-
218     setof(X, member(X, Lst), Set),
219     length(Lst, N),
220     length(Set, N).
221
222 output_data(File,L):-
223         open(File,'write',S),
224         (foreach(X,L),
225          param(S) do
226            writeq(S,X),writeln('.')
227         ),
228         close(S).
229
230 deleteLastElement([_], []).
231 deleteLastElement([Head], [Head|NTail]):-
232   deleteLastElement(Tail, NTail).
233
234 del(_, [], []).
235 del(X, [X|L1], L2):-
236      del(X,L1,L2).
237  del(X, [H|L1], [H|L2]):-
238      del(X,L1,L2),
239      X\==H.
240
241 substring(X,S) :-
242   append(_,T,S) ,
243   append(X,_,T) ,
244   X \= [].
```

Listing B.1: Part of an agent program to plan service orchestration

Appendix C

BPMN specification and rules

C.1 BPMN specification

- $\mathbb{S}$ (or node(start) fluents) is a set of *start events*
- $\mathbb{E}$ (or node(end) fluents) is a set of *end events*
- $\mathbb{T}$ (or task(_,_,_) fluents) is a set of *service activities* ; a task $T \in \mathbb{T}$ is a finite process with a set of inputs and a set of outputs, to be executed within a finite interval of time.
- $\mathbb{F}$ (or flow(_,_) fluents) is a set of *workflow links* , $\mathbb{F} \subseteq \mathbb{O} \times \mathbb{O}$, where $\mathbb{O} = \mathbb{S} \cup \mathbb{E} \cup \mathbb{T} \cup \mathbb{G} \cup \mathbb{M}$ is the join set of objects. All the component sets are pairwise disjoint. A sequence flow is used to link two entities of event, activity or gateway in a process diagram and specify a control flow relation. In addition, a sequence flow determines a enabling of entity in a sequence flow after the completion of a preceding entity in the same flow.
- $\mathbb{G}$ is a set of *split gateways*, where branching of the workflow takes place; two disjoint subtypes of splits are considered:
  - $\mathbb{GX}$ (or gateway(xorS) fluents) is a set of *exclusive splits* where one and only one of the alternative paths can be followed (a split of XOR type). In other words, $\mathbb{GX}$ allows a flow to be split into two or more flows when the incoming flow is enabled, the gateway is passed to one of the outgoing flows based on a specified condition that can select one of the outgoing flows.
  - $\mathbb{GP}$ (or gateway(andS) fluents) is a set of *parallel splits* where all the paths

of the workflow are to be followed (a split of AND type) In other words, $\mathbb{GP}$ allows a single flow to be split into two or more branches which can execute tasks concurrently.

- $\mathbb{M}$ is a set of *merge gateways* where two or more paths meet; two further disjoint subtypes of merge modes are considered:
  - $\mathbb{MX}$ (or gateway(xorJ) fluents) is a set of *exclusive merge* nodes where one and only one input path is taken into account (a merge of XOR type). In other words, $\mathbb{MX}$ allows two or more flows to be join into a single subsequence flow such that each enabling of an incoming flow results in the subsequent flow.
  - $\mathbb{MP}$ (or gateway(andJ) fluents) is a set of *parallel merge* nodes where all the paths are combined together (a merge of AND type). In other words, $\mathbb{MP}$ allows two or more parallel flows to be joined into a single subsequent flow when all input flows have been enabled.

- $\mathbb{GE}$ (or guard(_,_,_) fluents) is a set of *guard condition expressions* where the splits and merges depend on logical guard conditions assigned to particular branches. It is assumed that there is a defined partial function Con: $\mathbb{F} \rightarrow \mathbb{GE}$ assigning logical formulas to links. In particular, the function is defined for links belonging to $\mathbb{G} \times \mathbb{O} \cup \mathbb{O} \times \mathbb{M}$, i.e. outgoing links of split nodes and incoming links of merge nodes. The conditions are responsible for workflow control.

C.2 BPMN Well-formed

Having selected the core BPMN elements it is necessary to state restrictions on the overall diagram structure. The following is a set of typical requirements defining the so-called *well-formed diagram* [OMG].

- $\forall s \in \mathbb{S}, in(s) = \emptyset$ and $|out(s)| = 1$ is any start node $s \in \mathbb{S}$ has no incoming links and exactly one outgoing link,
- $\forall 3 \in \mathbb{E}, in(e) = 1$ and $|out(e)| = \emptyset$ is any end event node $e \in \mathbb{E}$ has no outgoing links and exactly one incoming link,
- $\forall T \in \mathbb{T}, in(s) = 1$ and $|out(T)| = 1$ is any task node $T \in \mathbb{T}$ has exactly one input and one output link,

- $\forall g \in \mathbb{G}, in(g) = 1$ and $|out(g)| \geq 2$ is any split node $g \in \mathbb{G}$ has exactly one incoming link and at least two outgoing ones,
- $\forall m \in \mathbb{M}, in(m) \geq 2$ and $|out(m)| = 1$ is any merge node $m \in \mathbb{M}$ has at least two incoming links and exactly one outgoing link,
- $\forall f \in \mathbb{F}, f \in out(\mathbb{S} \cup \mathbb{T} \cup \mathbb{G} \cup \mathbb{M}) \times in(\mathbb{E} \cup \mathbb{T} \cup \mathbb{G} \cup \mathbb{M})$ every link joins some legal output of some object with a legal input of some (other) objects.
- $\forall GE \in \mathbb{GE}$, $\mathbb{GE} \subseteq \mathbb{P} \times \Sigma \times \mathbb{V}$ where $\mathbb{P}$ = a set of parameters, $\Sigma = \{\wedge, \vee, =, \neq, \geq, \leq, >, <\}$ and $\mathbb{V}$ = a set of values
- every object $o \in \mathbb{O}$ is on some path from some start event and an end event.

C.3 Well-formed BPMN rules

1. A start event must be a source of a sequence.

```
rule1(Z)  :-
holds(flow(start,_),Z).
```

2. There must not exist a connection between a start and an end event.

```
rule2(Z) :-
\+ (knows_val([End],node(End),Z), End \= start,
knows_val([Op],flow(Op,End),Z), holds(flow(start, Op),Z)),
\+ (knows_val([End],node(End),Z), End \= start,
knows_val([Op],flow(Op,End),Z), holds(flow(Op, start),Z)).
```

3. A connection between two stops events is not allowed.

```
rule3(Z) :-
\+ (knows_val([Y1],node(Y1),Z),knows_val([Y2],node(Y2),Z),
Y1\==Y2, Y1 \= start, Y2 \= start, holds(flow(Y1,Y2),Z)),
\+ (knows_val([Y1],node(Y1),Z),knows_val([Y2],node(Y2),Z),
Y1\==Y2, Y1 \= start, Y2 \= start, holds(flow(Y2,Y1),Z)).
```

4. An end event must be a target of a sequence.

```
rule4(Z)  :-
\+ (knows_val([X],node(X),Z), X \= start,
knows_val([Y],flow(Y,X),Z), not_holds(flow(Y,X),Z)).
```

5. An end event must not be a source of a sequence flow, there must not be an outgoing sequence flow.

```
rule5(Z)  :-
\+ (knows_val([X],end_node(X),Z), X \= start, holds(flow(X,_),Z)).
```

6. Any split node has exactly one incoming link and at least two outgoing ones.

```
rule6(Z)  :-
\+ (knows_val([NSplit],gateway(NSplit),Z),
(knows_val([X1],flow(NSplit,X1),Z);
knows_val([],flow(NSplit,X1,_),Z)),
(knows_val([X2],flow(NSplit,X2),Z);
knows_val([X2],flow(NSplit,X2,_),Z)),
X1 \= X2, not_holds(flow(_,NSplit),Z)).
```

7. Any merge node has at least two incoming links and exactly one outgoing link.

```
rule7(Z)  :-
\+ (knows_val([NMerge],gateway(NMerge),Z),
knows_val([X1],flow(X1,NMerge),Z),
knows_val([X2],flow(X2,NMerge),Z), X1 \= X2,
not_holds(flow(NMerge,_),Z)).
```

8. The following is a rule for checking correctness of guard condition expression.

```
check_guard_expression(Z) :-
\+ (holds(guard(GE),Z),not_holds(flowPath(_,GE),Z)).
```

Bibliography

[ACD+05] V. Agarwal, G. Chafle, K. Dasgupta, N. M. Karnik, A. Kumar, S. Mittal, and B. Srivastava. Synthy: A system for end to end composition of web services. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(4):311–339, 2005.

[ACKM04] G. Alonso, F. Casati, H. A. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer, 2004.

[AEE06] A. Alamri, M. A. Eid, and A. El-Saddik. Classification of the state-of-the-art dynamic web services composition techniques. *International Journal of Web and Grid Services*, 2(2):148–166, 2006.

[And10] D. Androcec. Simulating bpmn models with prolog. In *Central European Conference on Information and Intelligent Systems*, 2010.

[BCG+05] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic service composition based on behavioral descriptions. *International Journal of Cooperative Information Systems*, 14(4):333–376, 2005.

[BDFR03] B. Benatallah, M. Dumas, M. C. Fauvet, and F. A. Rabhi. *Patterns and skeletons for parallel and distributed computing*. Towards patterns of web services composition, Springer, 2003.

[BEB98] A. Bouguettaya, A. K. Elmagarmid, and B. Benatallah. *Interconnecting heterogeneous information systems*. Kluwer Academic Publishers, 1998.

[BMB+00] B. Benatallah, B. Medjahed, A. Bouguettaya, A. K. Elmagarmid, and J. Beard. Composing and maintaining web-based virtual enterprises. In *TES*, pages 155–174, 2000.

[BP14] G. Baryannis and D. Plexousakis. Fluent calculus-based semantic web service composition and verification using wssl. In *Service-Oriented Computing - ICSOC 2013 Workshops*, volume 8377 of *Lecture Notes in Computer Science*, pages 256–270. Springer International Publishing, 2014.

[BPT10] P. Bertoli, M. Pistore, and P. Traverso. Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence*, 174(3-4):316–361, 2010.

[CC14] I. B. Caicedo-Castro. A Desription-based Service Search System. In *PhD Symposium, in conjonction with the 12th International Conference on Service Oriented Computing (ICSOC'14)*, Paris, France, novemeber 2014. 6 pages.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web service description language (wsdl) 1.1. W3C note, W3C, Mar 2001. Web 12 Jan. 2015, http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[CGLN01] D. Calvanese, G. B. Giacomo, M. Lenzerini, and D. Nardi. Reasoning in expressive description logics. *Handbook of Automated Reasoning*, 2:1581–1634, 2001.

[CMRW07] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language. W3C recommendation, W3C, June 2007. Web 12 Jan. 2015, http://www.w3.org/TR/2007/REC-wsdl20-20070626/.

[CRB04] A. Cimatti, M. Roveri, and P. Bertoli. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 159(1-2):127–206, 2004.

[CS01] F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–163, 2001.

[CSHG09] V. R. Chifu, I. Salomie, I. Harsa, and M. Gherga. Semantic web service composition method based on fluent calculus. In *11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 325–332, Timisoara, Romania, 2009.

[Dav93] T. H. Davenport. *Process Innovation: Reengineering Work through Information Technology.* Harvard Business School Press, Boston, MA, USA, 1993.

[DDDGB08] G. Decker, R. Dijkman, M. Dumas, and L. García-Bañuelos. Transforming bpmn diagrams into yawl nets. In *Business Process Management*, pages 386–389. Springer, 2008.

[DDO08] R. M. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in bpmn. *Information and Software Technology*, 50(12):1281–1294, 2008.

[Dog98] A. Dogac. A survey of the current state-of-the-art in electronic commerce and research issues in enabling technologies. In *Euro-Med Net 98 Conference, Electronic Commerce Track, Nicosia, Cyprus*, 1998.

[DPAM07] L. A. Digiampietri, J. Pérez-Alcázar, and C. B. Medeiros. Ai planning in web services composition: a review of current approaches and a new solution. *VI ENIA - nos Anais do XXVII Congresso da Sociedade Brazileira de Computacao (CSBC2007)*, pages 983–992, July 2007.

[DS05] S. Dustdar and W. Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[DS12] Semantic Automated Discovery and Integration (SADI). Relationship to other web service standards, 2012. n.d. Web. 12 Jan. 2015, https://code.google.com/p/sadi/wiki/StandardsComparison.

[ECL] The eclipse constraint programming system. ECLiPSe Home, Cisco, n.d. Web. 24 Dec. 2014, http://eclipseclp.org/.

[Erl05] T. Erl. *Service-oriented architecture (SOA): concepts, technology, and design.* Prentice Hall, August 2005.

[ESB10] Expertos en java y gestion de proyectos j2ee/jee, 2010. n.d. Web. 12 Jan. 2015, http://www.consultoriajava.com.

[ESB14] N. El-Saber and A. Boronat. Bpmn formalization and verification using maude. In *Proceedings of the 2014 Workshop on Behaviour Modelling-Foundations and Applications*, page 1. ACM, 2014.

[FHH+01] D. Fensel, F. V. Harmelen, I. Horrocks, D. L. McGuinness, and P. F. Patel-Schneider. Oil: An ontology infrastructure for the semantic web. *IEEE intelligent systems*, 16(2):38–45, 2001.

[FL07] J. Farrell and H. Lausen. Semantic annotations for WSDL and XML schema. W3C recommendation, W3C, #aug# 2007. http://www.w3.org/TR/2007/REC-sawsdl-20070828/.

[FN72] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3):189–208, 1972.

[GL99] G. D. Giacomo and J. H. Levesque. An incremental interpreter for high-level programs with sensing. In *Logical Foundations for Cognitive Agents*, pages 86–102. Springer, 1999.

[GT00] F. Giunchiglia and P. Traverso. Planning as model checking. In *Recent Advances in AI Planning*, pages 1–20. Springer, 2000.

[Hay71] P. J. Hayes. *The Frame Problem and Related Problems on Artificial Intelligence*. Stanford University, 1971.

[HS05] M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[KMCW05] R. Khalaf, N. Mukhi, F. Curbera, and S. Weerawarana. The business process execution language for web services. In *Process-Aware Information Systems: Bridging People and Software Through Process Technology*. John Wiley & Sons, 2005.

[KS89] R. Kowalski and M. Sergot. A logic-based calculus of events. In *Foundations of knowledge base management*, pages 23–55. Springer, 1989.

[Lak99] G. Lakemeyer. On sensing and off-line interpreting in golog. In *Logical Foundations for Cognitive Agents*, pages 173–189. Springer, 1999.

[Lba05] A. Lbath. Method and device for automatic production of context aware mobile services. In *PATENTSCOPE*. No WO 006721, 2005.

[Lin08] F. Lin. Situation calculus. *Handbook of knowledge representation*, 3:3–88, 2008.

[LLCC+13] P. N. Lumpoon, M. Lei, I. B. Caicedo-Castro, M. C. Fauvet, and A. Lbath. Context-aware service discovering system for nomad users. In *7th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)*, Chiang Mai, Thailand, 2013.

[LLK+11] P. N. Lumpoon, M. Lei, T. Kamnardsiri, M. C. Fauvet, and A. Lbath. Illustrating some issues raised when designing context-aware personalized services for mobile users. In *6th International Conference on Software, Knowledge, Information Management and Applications (SKIMA 2012)*, 2011.

[LOKX13] Zheng Li, Liam O'Brien, Jacky Keung, and Xiwei Xu. Effort-oriented classification matrix of web service composition. *CoRR Computing Research Respository*, abs/1302.2201, 2013.

[LP14] A. Ligoza and T. Potempa. Ai approach to formal analysis of bpmn models: Towards a logical model for bpmn diagrams. In *Advances in Business ICT*, pages 69–88. Springer, 2014.

[LRL+97] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1-3):59–83, 1997.

[MBB+03] B. Medjahed, B. Benatallah, A. Bouguettaya, A. H. Ngu, and A. K. Elmagarmid. Business-to-business interactions: issues and enabling technologies. *The VLDB Journal The International Journal on Very Large Data Bases*, 12(1):59–85, 2003.

[MBH+04] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. Owl-s: Semantic markup for web services, 2004.

[McD00] D. V. McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2):35–55, 2000.

[MGH+98] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl-the planning domain definition language. *AI Magazine*, 1998.

[MM04] N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[MMV+05] R. Mulye, J. Miller, K. Verma, K. Gomadam, and A. Sheth. A semantic template based designer for web processes. In *Proceedings IEEE International Conference on Web Service*. IEEE, 2005.

[MPT08] A. Marconi, M. Pistore, and P. Traverso. Automated composition of web services: the ASTRO approach. *IEEE Data Engineering Bulletin Issues*, 31(3):23–26, 2008.

[MS02] S. A. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, editors, *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 482–496, Toulouse, France, 2002. Morgan Kaufmann.

[MSZ01] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[NAI+03] D. S. Nau, T. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: an HTN planning system. *Journal of Artificial Intelligence Research JAIR*, 20:379–404, 2003.

[Nic05] D. Nickull. Service oriented architecture. Technical report, White Paper, Adobe Systems, Inc, May 2005.

[OMG] OMG. Business process model and notation (bpmn)) version 2.0.2. n.d. Web. 20 Dec. 2014, http://www.omg.org/spec/BPMN/2.0.2/.

[Ped94] E. Pednault. Adl and the state-transition model of action. *Journal of Logic and Computation*, 4(5):467–512, 1994.

[Pee05] J. Peer. Web service composition as ai planning-a survey. *University of St. Gallen*, 2005.

[POSV04] A. A. Patil, S. A. Oundhakar, A. P. Sheth, and K. Verma. Meteor-s web service annotation framework. In *Proceedings of the 13th international conference on World Wide Web*, pages 553–562. ACM, 2004.

[PvdH07] M. P. Papazoglou and W. J. van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *VLDB Very Large Database Journal*, 16(3):389–415, 2007.

[PW06] F. Puhlmann and M. Weske. *Investigations on soundness regarding lazy activities*. Springer, 2006.

[Rei01] R. Reiter. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*, volume 16. MIT press Cambridge, 2001.

[RN95] S. J. Russell and P. Norvig. *Artificial intelligence - a modern approach: the intelligent agent book*. Prentice Hall series in artificial intelligence. Prentice Hall, 1995.

[RS04] J. Rao and X. Su. A survey of automated web service composition methods. In J. Cardoso and A. P. Sheth, editors, *Semantic Web Services and Web Process Composition, First International Workshop, SWSWPC*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54, San Diego, CA, USA, 2004. Springer.

[SBe00] E. Lindencrona S. Brinkkemper and A. Solvberg (eds.). *The B2B E-commerce Revolution: Convergence, Chaos, and Holistic Computing, in Information System Engineering: State of the Art and Research Themes*. Springer-Verlag Ltd., London., 2000.

[SK03] B. Srivastava and J. Koehler. Web service composition-current solutions and open problems. In *ICAPS International Conference on Automated Planning & Scheduling workshop on Planning for Web Services*, volume 35, pages 28–35, 2003.

[SPW+04] E. Sirin, B. Parsia, D. Wu, J. A. Hendler, and D. S. Nau. HTN planning for web service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377–396, 2004.

[SQV+14] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu. Web services composition: A decade's overview. *Information Sciences*, 280:218–238, 2014.

[TDE02] S. Thöne, R. Depke, and G. Engels. Process-oriented, flexible composition of web services with UML. In S. Spaccapietra, S. T. March, and Y. Kambayashi, editors, *Conceptual Modeling - ER 2002, 21st International Conference on Conceptual Modeling*, volume 2503 of *Lecture Notes in Computer Science*, pages 390–401, Tampere, Finland, 2002. Springer.

[Tea] Activiti Team. Activiti bpm platform. Activiti. Alfresco, n.d. Web. 03 Mar. 2015, http://activiti.org/.

[Thi98] M. Thielscher. Introduction to the fluent calculus. *Electronic Transactions on Artificial Intelligence*, 2:179–192, 1998.

[Thi02] M. Thielscher. Alan: Designing an axiomatisation language for autonomous agents and mobile robots, 2002. March 2002, Web. 20 Dec. 2014, http://www.computationallogic.org/content/projects/alan.php.

[Thi05] M. Thielscher. Flux: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4-5):533–565, 2005.

[WGHS99] M. Weske, T. Goesmann, R. Holten, and R. Striemer. *A reference model for workflow application development processes*, volume 24. ACM, 1999.

[WNI+09] S. M. Watt, V. Negru, T. Ida, T. Jebelean, D. Petcu, and D. Zaharie, editors. *11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2009, Timisoara, Romania, September 26-29, 2009*. IEEE Computer Society, 2009.