Chapter 17
Adaptive Composition and QoS Optimization of Conversational Services Through Graph Planning Encoding

Min Chen, Pascal Poizat and Yuhong Yan

Abstract Service-Oriented Computing supports description, publication, discovery, composition of services as well as QoS optimization of service composition to fulfil end-user needs. Yet, service composition processes commonly assume that service descriptions and user needs share the same abstraction level, and that services have been pre-designed to integrate. To release these strong assumptions and to augment the possibilities of composition, we add adaptation features into the service composition process using semantic structures for exchanged data, for service functionalities, and for user needs. Graph planning encodings enable us to retrieve service compositions efficiently. Our composition technique supports conversations for both services and user needs, and it is fully automated and can interact with state-of-the-art graph planning tools. In addition to service composition, QoS optimization aims at satisfying end-user needs about quality requirements. However, most existing work on QoS optimization is studied on the assumption that services are stateless. To obtain a solution with the best QoS value, we propose a QoS-aware service composition method to achieve QoS optimization during the adaptive composition over conversational services. An example is given as a preliminary proof of our QoS-aware service composition method.

M. Chen (✉) · Y. Yan
Concordia University, Montreal, Canada
e-mail: minchen2008halifax@yahoo.com

Y. Yan
e-mail: yuhong@encs.concordia.ca

P. Poizat
Université Nanterre Paris Ouest La Défense, Nanterre, France
e-mail: pascal.poizat@lri.fr

P. Poizat
LIP6 UMR 7606 CNRS, Orsay, France

17.1 Introduction

Task-Oriented Computing envisions a user-friendly pervasive world where *user tasks* corresponding to a (potentially mobile) user would be achieved by the automatic assembly of resources available in her/his environment. Service-Oriented Computing [28] (SOC) is a cornerstone towards the realization of this vision, through the abstraction of heterogeneous resources as services and automated composition techniques [17, 22, 30]. However, services being elements of composition developed by different third-parties, their reuse and assembly naturally raises composition mismatch issues [2, 13]. Moreover, Task-Oriented Computing yields a higher description level for the composition requirements, i.e., the user task(s), as the user only has an abstract vision of her/his needs which are usually not described at the service level. These two dimensions of interoperability, namely *horizontal* (communication protocol and data flow between services) and *vertical matching* (correspondence between an abstract user task and concrete service capabilities) should be supported in the composition process.

Software adaptation is a promising technique to augment component reusability and composition possibilities, thanks to the automatic generation of software pieces, called adaptors, solving mismatch out in a non-intrusive way [31]. More recently, adaptation has been applied in SOC to solve mismatch between services and clients (e.g., orchestrators) [12, 23, 27]. In this article we propose to add adaptation features in the service composition process itself. More precisely, we propose an automatic composition technique based on *planning*, a technique which is increasingly applied in SOC [15, 29] as it supports automatic service composition from underspecified requirements, e.g., the data one requires and the data one agrees to give for this, or a set of capabilities one is searching for. Such requirements do not refer to service operations or to the order in which they should be called, which would be ill-suited to end-user composition.

In addition to service composition, QoS optimization aims at satisfying end-user needs about quality requirements. To obtain the solution with the best QoS values for a service composition process, we propose a QoS-aware service composition method to realize QoS optimization during the adaptive service composition.

Outline. Preliminaries on planning are given in Sect. 17.2. After introducing our formal models in Sect. 17.3, Sect. 17.4 presents our encoding of service composition into a planning problem, and Sect. 17.5 proposes a QoS-aware service composition method over conversational services through planning graph as an extension work. Related work is discussed in Sect. 17.6 and we end with conclusions and perspectives.

17.2 Preliminaries

In this section we give a short introduction to AI planning [18].

Definition 17.1. Given a finite set $L = \{p_1, \ldots, p_n\}$ of proposition symbols, a *planning problem* [18] is a triple $P = ((S, A, \gamma), s_0, g)$, where:

- $S \subseteq 2^L$ is a set of states.
- A is a set of actions, an *action* a being a triple $(pre, effect^-, effect^+)$ where $pre(a)$ denotes the preconditions of a, and $effect^-(a)$ and $effect^+(a)$, with $effect^-(a) \cap effect^+(a) = \emptyset$, denote respectively the negative and the positive effects of a.
- γ is a state transition function such that, for any state s where $pre(a) \subseteq s$, $\gamma(s, a) = (s - effect^-(a)) \cup effect^+(a)$.
- $s_0 \in S$ and $g \subseteq L$ are respectively the initial state and the *goal*.

Two actions a and b are *independent* iff $effect^-(a) \cap [pre(b) \cup effect^+(b)] = \emptyset$ and $effect^-(b) \cap [pre(a) \cup effect^+(a)] = \emptyset$. An action set is independent when its actions are pairwise independent. A *plan* is a sequence of actions $\pi = a_1; \ldots; a_k$ such that $\exists s_1, \ldots, s_k \in S,\ s_1 = s_0,\ \forall i \in [1, k],\ pre(a_i) \in s_{i-1} \wedge \gamma(s_{i-1}, a_i) = s_i$. The definition in [18] takes into account predicates and constant symbols which are then used to define states (ground atoms made with predicates and constants). We directly use propositions here.

Graph Planning [7] is a technique that yields a compact representation of relations between actions and represent the whole problem world. A planning graph $G = (V, E)$ is a directed acyclic leveled graph. It has two kinds of vertices $V = V_A \cup V_P$ where V_A is the vertices representing actions and V_P representing propositions. And edges $E = (V_P \times V_A) \cup (V_A \times V_P)$ connect the vertices. The levels alternate proposition levels P_i and action levels A_i. The initial proposition level P_0 contains the initial propositions (s_0). The planning graph is constructed from P_0 using a polynomial algorithm. An action a is put in layer A_i iff $pre(a) \subseteq P_{i-1}$ and then $effect^+(a) \subseteq P_i$. Specific actions (no-ops) are used to keep data from one layer to the next one, and arcs to relate actions with used data and produced effects. Graph planning also introduces the concept of mutual exclusion (mutex) between non independent actions. Mutual exclusion is reported from a layer to the next one while building the graph. The planning graph actually explores multiple search paths at the same time when expanding the graph, which stops at a layer A_k iff the goal is reached ($g \subseteq A_k$) or in case of a fixpoint ($A_k = A_{k-1}$). In the former case there exists at least a solution, while in the later there is not. Solution(s) can be obtained using backward search from the goal. Planning graphs whose computation has stopped at level k enable to retrieve all solutions up to this level. Additionally, planning graphs enable to retrieve solutions in a concise form, taking benefit of actions that can be done in parallel (denoted ||).

An example is given in Fig. 17.1 where we suppose the initial state is {a} and the objective is {e}. Applying U in the first action layer, for example, is possible because a is present; and this produces b and c. The extraction of plans from the graph is performed using a backward chaining technique over action layers, from the final state (objective) back to the initial one. In the example, plans U;Y, Z;Y, (U||Z);Y and (U||Z);S can be obtained (see bold arcs in Fig. 17.1 for U;Y). However, U and Z are in mutual exclusion. Accordingly, since there is no other way to obtain c and d

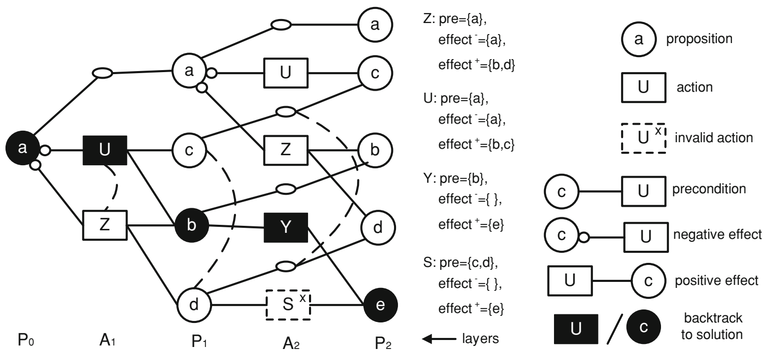
Z: pre={a}, effect⁻={a}, effect⁺={b,d}
U: pre={a}, effect⁻={a}, effect⁺={b,c}
Y: pre={b}, effect⁻={ }, effect⁺={e}
S: pre={c,d}, effect⁻={ }, effect⁺={e}
proposition
action
invalid action
precondition
negative effect
positive effect
backtrack to solution
layers
P0
A1
P1
A2
P2

Fig. 17.1 Graphplan example

than with exclusive actions, these two facts are in exclusion in the next (fact) layer, making S impossible. Note that other nodes are indeed in mutual exclusion (such as U and Z in A_1, or two no-ops in A_2 but we have not represented this for clarity).

17.3 Modeling

In this section, we present our formal models, grounding service composition. Table 17.1 lists the symbols used in this section. Both services and composition requirements support conversations. Therefore, we begin with their definition. We then present the structures supporting the definition of semantic data and capabilities. Finally, we present models for services and service composition requirement.

17.3.1 Conversation Modeling

Different models have been proposed to support service discovery, verification, testing, composition or adaptation in presence of service conversations [3, 9, 23]. They

Table 17.1 Summary of symbols

Symbol	Definition	Symbol	Definition
WF^X	A Workflow (WF) over a set of names X	P_A	Activities
P_{so}	XOR-Splits	P_{sa}	AND-Splits
P_{jo}	OR-Joins	P_{ja}	AND-Joins
$\mathcal{D}$	Data Semantic Structure (DSS)	$\mathcal{K}$	Capability Semantic Structure (CSS)
O	A set of operations	W	A set of services

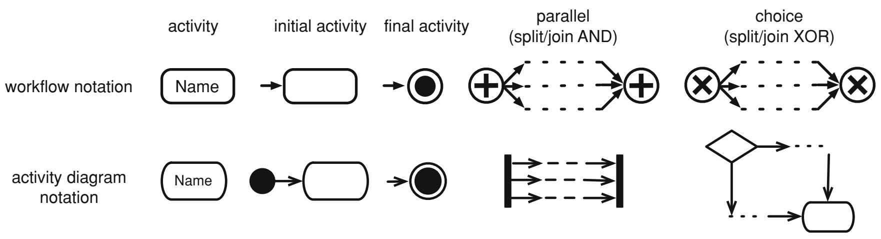

Fig. 17.2 Workflow notation and relation to the UML activity diagrams

mainly differ in their formal grounding (Petri nets, transition systems, or process algebra), and the subset of service languages being supported. Since we target centralized composition (orchestration) with possible parallel service invocation, we choose the workflow model from [19]. An important benefit of workflow models is that they can be related via model transformation to graphical notations that are well-known by the software engineers, e.g., UML activity diagrams (Fig. 17.2) or BPMN. Additionally, workflows are more easily mastered by a non-specialist through pre-defined patterns (sequence, alternative choice, parallel tasks). Transition systems models could yield a simpler encoding as a planning problem but raise issues when it comes to implement the composition models, requiring model filtering to remove parts in the composition models which are not implementable in the target language [23].

Definition 17.2. Given a set of activity names N, a Workflow (WF) [19] is a tuple $WF^N = (P, \rightarrow, Name)$. P is a set of process elements (or workflow nodes) which can be further divided into disjoint sets $P = P_A \cup P_{so} \cup P_{sa} \cup P_{jo} \cup P_{ja}$, where P_A are activities, P_{so} are XOR-Splits, P_{sa} are AND-splits, P_{jo} are OR-Joins, and P_{ja} are AND-Joins. $\rightarrow \subseteq P \times P$ denotes the control flow between nodes. $Name : P_A \rightarrow N$ is a function assigning activity names to activity nodes.

We note $\bullet x = \{y \in P | y \rightarrow x\}$ and $x\bullet = \{y \in P | x \rightarrow y\}$. We require that WF are well-structured [19] and without loop. A significant feature of well-structured workflows is that the XOR-splits and the OR-Joins, and the AND-splits and the AND-splits appear in pairs (Fig. 17.2). Moreover, we require $|\bullet x| \leq 1$ for each x in $P_A \cup P_{sa} \cup P_{so}$ and $|x \bullet| \leq 1$ for each x in $P_A \cup P_{ja} \cup P_{jo}$.

17.3.2 Semantic Structures

In our work we use semantic information to enrich the service composition process and its automation. We have two kinds of semantic information. Capabilities represent the functionalities that are either requested by the end-users or provided by services. They are modelled using a Capability Semantic Structure (CSS). Further, service inputs and outputs are annotated using a Data Semantic Structure (DSS).

Table 17.2 eTablet buying—DSS relations: $d_1 \sqsubset d_2$ (left), $d_1 \lhd_x d_2$ (right)

d_1	d_2
etablet	pear_product
etelephone	pear_product
pear_product	product
product_price	order_amount
user_address	shipping_addr
user_address	billing_addr
user_address	address

d_1	x	d_2
pear_product_info	price	product_price
pear_product_info	details	product_technical_information
user_info	name	user_name
user_info	address	user_address
user_info	cc	credit_card_info
user_info	pim	pim_wallet
pim_wallet	paypal	paypal_info
pim_wallet	amazon	amazon_info
paypal_info	login	paypal_login
paypal_info	pwd	paypal_pwd
amazon_info	login	amazon_login
amazon_info	pwd	amazon_pwd
credit_card_info	number	credit_card_number
credit_card_info	name	credit_card_holder_name

We define a *Data Semantic Structure (DSS)* as a tuple $(\mathcal{D}, \lhd, \sqsubset)$ where $\mathcal{D}$ is a set of concepts (or semantic data type[1]) that represent the semantics of some data, $\lhd$ is a composition relation ($(d_1, x, d_2) \in \lhd$, also noted $d_1 \lhd_x d_2$ or simply $d_1 \lhd d_2$ when x is not relevant for the context, means a d_1 is composed of an x of type d_2), and $\sqsubset$ is a subtyping relation ($d_1 \sqsubset d_2$ means d_1 can be used as a d_2). We require there is no circular composition. DSSs are the support for the automatic decomposition (of d into D if $D = \{d_i \mid d \lhd d_i\}$), composition (of D into d if $D = \{d_i \mid d \lhd d_i\}$) and casting (of d_1 into d_2 if $d_1 \sqsubset d_2$) of data types exchanged between services and orchestrator. We also define a *Capability Semantic Structure (CSS)* as a set $\mathcal{K}$ of concepts that correspond to capabilities.

Application. We will illustrate our composition technique on a simple, yet realistic, case study: the online buying of an eTablet. A DSS describes concepts and relations for this case study. For place matters, we only give the relations here (Table 17.2) since concepts can be inferred from these and from the service operation signatures, below.

[1] In this paper, the concepts of semantics and type of data are unified.

Table 17.3 eTablet buying—services' operations

Service	Operation	Profile
w_1	order	pear_product → pear_product_info, as_sessionid :: product_selection : 20
w_1	cancel	as_sessionid → ∅ :: nil:2
w_1	ship	shipping_addr, as_sessionid → ∅ :: shipping_setup:10
w_1	bill	billing_addr, as_sessionid → ∅ :: billing_setup:25
w_1	charge	credit_card_info, as_sessionid → ∅ :: payment:10
w_1	gift_wrapper	giftcode, as_sessionid → ∅ :: payment:20
w_1	ack	as_sessionid → tracking_num :: order_finalization:5
w_2	order	product → e_sessionid :: product_selection:5
w_2	ship	shipping_addr, e_sessionid → order_amount :: shipping_setup:7
w_2	charge_pp	paypal_trans_id, e_sessionid → ∅ :: nil:12
w_2	charge_cc	credit_card_info, e_sessionid → ∅ :: payment:15
w_2	bill	billing_addr, e_sessionid → ∅ :: billing_setup:8
w_2	finalize	e_sessionid → tracking_num :: order_finalization:6
w_3	login	paypal_login, paypal_pwd → p_sessionid :: nil:10
w_3	get_credit	order_amount, p_sessionid → paypal_trans_id :: payment:20
w_3	ask_bill	address, p_sessionid → ∅ :: billing_setup:8
w_3	logout	p_sessionid → ∅ :: nil:4

17.3.3 Services

A service is a set of operations described in terms of capabilities, inputs, outputs and quality. Additionally, services have a conversation.

Definition 17.3. Given a CSS $\mathcal{K}$ and a DSS $\mathcal{D} = (\mathcal{D}, \lhd, \sqsubset)$, a *service* is a tuple $w = (O, WF^O)$, where O is a set of operations, an operation being a tuple (in, out, k, n) with $in \subseteq \mathcal{D}$, $out \subseteq \mathcal{D}$, $k \in \mathcal{K}$, n is the quality value, and WF^O is a workflow built over O.

For a simple service (without a conversation) w, a trivial conversation can be obtained with a workflow where $P_A = O(w)$ (one activity for each operation), $P_{so} = \{\otimes\}$, $P_{jo} = \{\overline{\otimes}\}$, $P_{sa} = P_{ja} = \emptyset$, and $\forall o \in P_A$, $\{(\otimes, o), (o, \overline{\otimes})\} \subseteq \rightarrow$. This corresponds to a generalized choice between all possible operations. An operation may not have a capability and the quality of the operation may not be given (we then let $k = \mathsf{nil}$). $o = (in, out, k, n)$ is also noted $o : in \rightarrow out :: k : n$. If several quality values are given for operation o, we calculate the aggregated QoS value n as an overall quality for o. How to calculate the aggregated quality value will be introduced in Sect. 17.5.1.

Application. To fulfill the user need, we have three services: pear_store (w_1, online store for pear products), ebay (w_2, general online shop) and paypal (w_3, online payment facilities). Their operations are given in Table 17.3 and their workflows are given in Fig. 17.3. The QoS of each operation in services is supposed to be throughput.

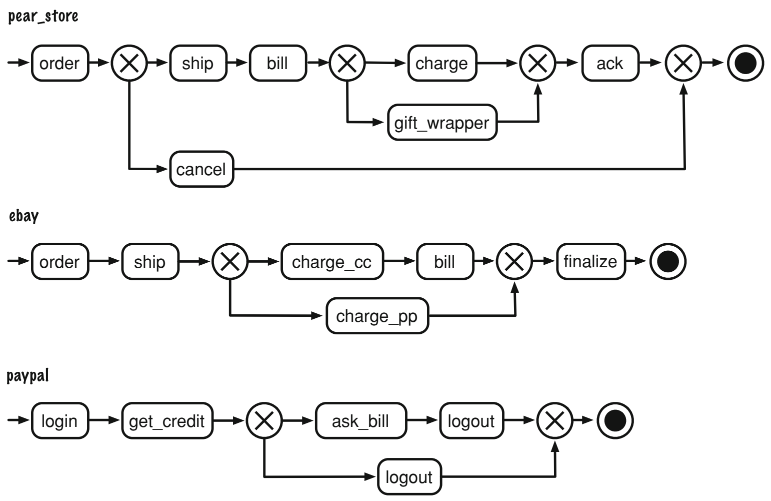

Fig. 17.3 eTablet buying—services' workflows

17.3.4 *Composition Requirements*

A service composition requirement is given in terms of the inputs the user is ready to provide and the outputs this user is expecting. Additionally, the capabilities that are expected from the composition are specified, and their expected ordering given under the form of a workflow.

Definition 17.4. Given a CSS $\mathcal{K}$ and a DSS $\mathcal{D} = (\mathcal{D}, \lhd, \sqsubset)$, a *composition requirement* is a tuple $(D_{\text{in}}, D_{\text{out}}, WF^{\mathcal{K}})$ where $D_{\text{in}} \subseteq \mathcal{D}$, $D_{\text{out}} \subseteq \mathcal{D}$, and $WF^{\mathcal{K}}$ is a workflow build over $\mathcal{K}$.

Application. The user requirement in our case study is $(\{etablet, user\_info\}, \{tracking\_num\}, wfc)$. As far as the wfc requirement workflow is concerned, we have two alternatives for it. The first one (Fig. 17.4, left) requires that payment is done after shipping and billing have been set up (which can be done in parallel). The second one (Fig. 17.4, right) is less strict and enables the payment to be done in parallel to shipping and billing setup.

17.4 Encoding Composition as a Planning Problem

In this section we present how service composition can be encoded as a graph planning problem. We will first explain how DSS can be encoded (to solve out horizontal adaptation). Then we will present how a generic workflow can be encoded. Based on

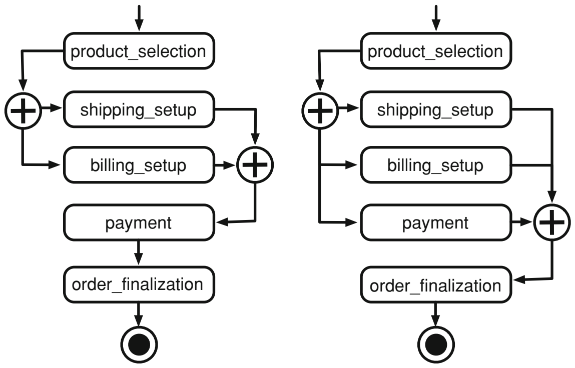

Fig. 17.4 eTablet buying—requirement workflows

this, we will then explain how services and composition requirements are encoded (the workflow of the later solving out vertical adaptation).

17.4.1 DSS Encoding

For each $d \lhd \{x_i : d_i\}$ in the DSS we have an action $comp_d(\bigcup_i \{d_i\}, \emptyset, \{d\})$ and an action $dec_d(\{d\}, \emptyset, \bigcup_i \{d_i\})$ to model possible (de)composition. Moreover, for each $d \sqsubset d'$ in the DSS we have an action $cast_{d,d'}(\{d\}, \emptyset, \{d'\})$ to model possible casting from d to d'.

17.4.2 Workflow Encoding

We reuse here a transformation from workflows to Petri net defined in [19]. Instead of mapping a workflow $(P, \rightarrow, Name)$ to a Petri net, we map it to a planning problem. Let us first define the set of propositions that are used. The behavioral constraints underlying the workflow semantics (e.g., an action being before/after another one) are supported through two kinds to propositions: $r_{x,y}$ and $c_{x,y}$. We also have a proposition I for initial states, and a proposition F for correct termination states. F will be used both for final states and for initial states (in this case to denote that a service can be unused). We may then define the actions that are used (Fig. 17.5):

- for each $x \in P_{sa}$, we have an action $a = \oplus x$ (Fig. 17.5a), for each $x \in P_{ja}$, we have an action $a = \bar{\oplus} x$ (Fig. 17.5b), and for each $x \in P_A$, we have an action $a = [Name(x)]x$ (Fig. 17.5c). In all three cases, we set $pre(a) = effect^-(a) = \bigcup_{y \in \bullet x} \{r_{x,y}\}$, and $effect^+(a) = \bigcup_{y \in x \bullet} \{c_{x,y}\}$.
- for each $x \in P_{so}$, for each $y \in x\bullet$, we have an action $a = \otimes x, y$ (Fig. 17.5d) and we set $pre(a) = effect^-(a) = \bigcup_{z \in \bullet x} \{r_{x,z}\}$, and $effect^+(a) = \{c_{x,y}\}$.

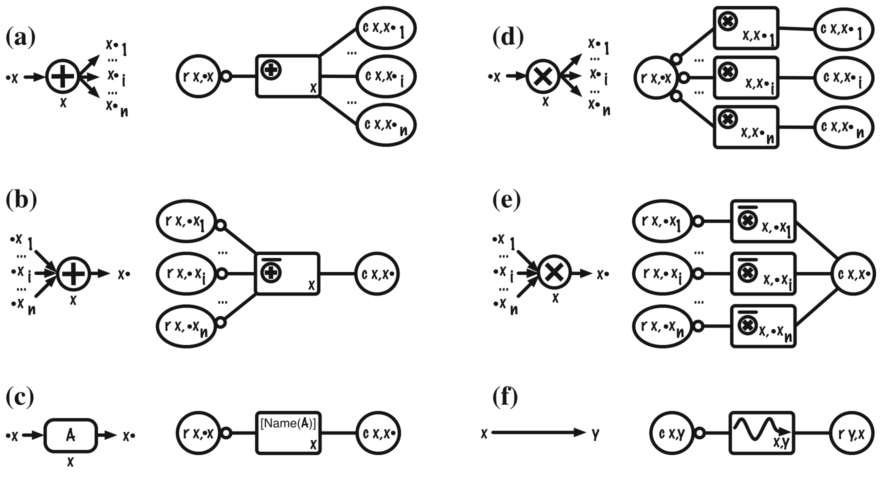

Fig. 17.5 Workflow encoding

- for each $x \in P_{jo}$, for each $y \in \bullet x$, we have an action $a = \bar{\otimes}x, y$ (Fig. 17.5e), and we set $pre(a) = effect^-(a) = r_{x,y}$, and $effect^+(a) = \bigcup_{z \in \bullet x}\{c_{x,z}\}$.
- for each $x \rightarrow y$, we have an action $a = \bar{\otimes}x, y$ (Fig. 17.5f) and we set $pre(a) = effect^-(a) = \{c_{x,y}\}$, and $effect^+(a) = \{r_{y,x}\}$.
- additionally, for any initial action a we add $\{I, F\}$ in $pre(a)$ and $effect^-(a)$.
- additionally, for any final action a we add $\{F\}$ in $effect^+(a)$.

17.4.3 Composition Requirements Encoding

A composition requirement $(D_{\text{in}}, D_{\text{out}}, WF^{\mathcal{K}})$ is encoded as follows. First we compute the set of actions resulting from the encoding of $WF^{\mathcal{K}}$ (see 17.4.2). Then we have to encode the fact that capabilities in the composition requirement encoding should interoperate with operations in service encodings. The idea is the following. Taking a service w, when a capability k is enabled at the current state of execution by $WF^{\mathcal{K}}$ then we should invoke an operation of capability k that is enabled at the current state by $WF^{O(w)}$ before any one of the capability possibly following k could be enabled. Moreover, an operation o with capability k of w can be invoked only iff this is enabled by the current state of execution in $WF^{O(w)}$ and k is enabled in $WF^{\mathcal{K}}$. To achieve this, we replace any action $a = [k]x$ in the encoding of $WF^{\mathcal{K}}$ by two actions, $a' = [k]x$ and $\overline{a'} = [k]\overline{x}$, and we set:

- $pre(a') = pre(a), effect^-(a') = effect^-(a), effect^+(a) = \{e_k, link_x\}$.
- $pre(\overline{a'}) = effect^-(\overline{a'}) = \{link_x, d_k\}, effect^+(\overline{a'}) = effect^+(a)$.

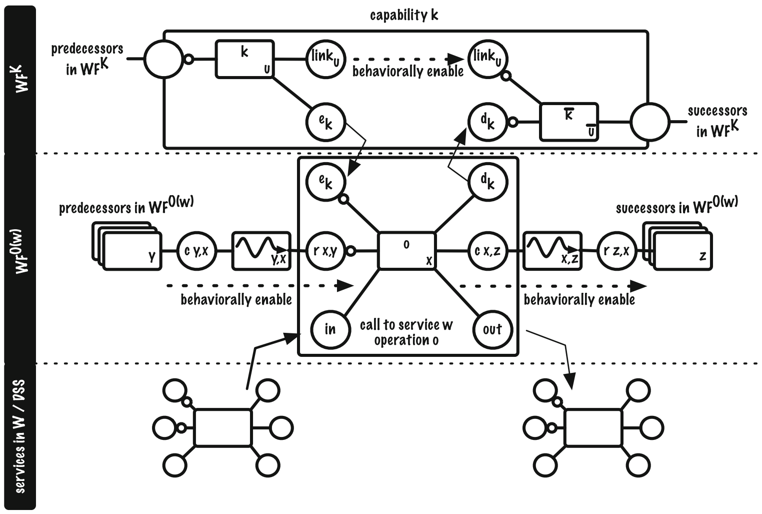

Fig. 17.6 Principle of interaction between service and requirement encodings

e_k and d_k enforce the synchronizing rules between capability workflow (defining when a capability k can be done) and service workflows (defining when an operation with capability k can be done) as presented in Fig. 17.6. $link_k$ ensure that two actions $a_1 = [k]x_1$ and $a_2 = [k]x_2$ with the same capability will not interact incorrectly when x_1 and x_2 are in parallel in a workflow.

17.4.4 Service Encoding

Each service $w = (O, WF^O)$ is encoded as follows. First we encode the workflow WF^O as presented in 17.4.2. Then, for each action $a = [o]x$ in this encoding we add:

- $in(o)$ in $pre(a)$ to model the inputs required by operation o and $out(o)$ in $effect^+(a)$ to model the outputs provided by operation o.
- $e_{k(o)}$ in $pre(a)$ and in $effect^-(a)$ and $d_{k(o)}$ in $effect^+(a)$ to implement the interaction with capabilities presented in 17.4.3 and in Fig. 17.6.

17.4.5 Overall Encoding

Given a DSS $\mathcal{D}$, a set of services W, and a composition requirement $(D_{\text{in}}, D_{\text{out}}, WF^{\mathcal{K}})$, we obtain the planning problem $((S, A, \gamma), s_0, g)$ as follows:

- $s_0 = D_{\text{in}} \cup \{wfc : I, wfc : F\} \bigcup_{w \in W} \{w : I, w : F\}$.
- $g = D_{\text{out}} \cup \{wfc : F\} \bigcup_{w? \in W} \{w : F\}$.
- $A = dss : ||\mathcal{D}|| \cup wfc : ||WF^{\mathcal{K}}|| \bigcup_{w \in W} w : ||WF^{O(w)}||$.
- S and γ are built with the rules in Definition 17.1.

where $||x||$ means the set of actions resulting from the encoding of x. Prefixing (denoted with $prefix$:) operates on actions and on workflow propositions (I, F, $r_{x,y}$, and $c_{x,y}$) coming from encodings. It is used to avoid name clashes between different subproblems. We suppose that, up to renaming, there is no service identified as dss or wfc.

17.4.6 Plan Implementation

Solving the planning problem, we may get a failure when there is no solution satisfying both that (i) a service composition exists to get D_{out} from D_{in}, (ii) using operations/capabilities in an ordering satisfying both used service conversations and capability conversation, (iii) leaving used services in their final state. In other cases, we obtain (see Sect. 17.2) a plan $\pi = L_1; \ldots; L_i; \ldots; L_n$ where ; is the sequence operator and each L_i is of the form $(P_{i,1}|| \ldots ||P_{i,j}|| \ldots ||P_{i,m_i})$ where || is the parallel operator and each $P_{i,j}$ is a workflow process element. First of all, we begin by filtering out π by removing from it all $P_{i,j}$ that is not of the form $dss : \ldots$ or $w : [o]x$, i.e., that is a purely structuring item, not corresponding to data transformation or service invocation. Given the filtered plan, we can generate a WS-BPEL implementation for it as done for transitions systems in [23]. Still, we may benefit here from the fact that actions that can be done in parallel are explicited in a graph planning plan (using operation ||), while in transition systems we only have interleaving semantics (finding out which actions can be done in parallel is much more complex). Therefore, for the main structure of the <process> ...< /process> element we replace the [23] state machine encoding by a more efficient version using sequence and flows. For π we get:

$$\langle \texttt{sequence} \rangle modeltrans(L_1) \ldots modeltrans(L_i) \ldots modeltrans(L-n) \langle \texttt{/sequence} \rangle$$

and for each $L_i = (P_{i,1}|| \ldots ||P_{i,j}|| \ldots ||P_{i,m_i})$ we have:

$$\langle \texttt{flow} \rangle \; modeltrans(P_{i,1}) \ldots modeltrans(P_{i,j}) \ldots modeltrans(P_{i,m_i})) \; \langle \texttt{/flow} \rangle$$

where $modeltrans$ is the transformation of basic assignment/communication activities defined in [23].

17.4.7 Tool Support

Our composition approach is supported with a tool, pycompose (Fig. 17.7), written in the Python language. This tool takes as input a DSS file, several service description files (list of operations and workflow), and the composition requirement (input

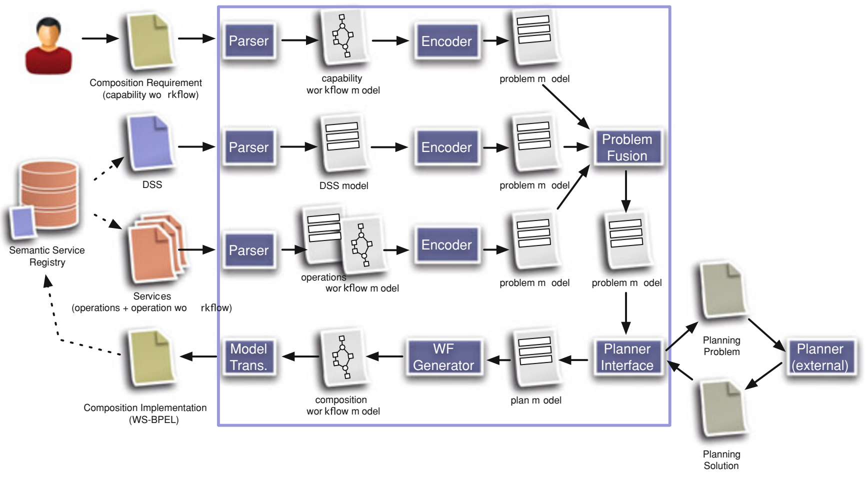

Fig. 17.7 Architecture of the pycompose tool

list, output list, and a workflow file). It then generates the encoding of this composition problem. pycompose supports through a command-line option the use of several planners: the original C implementation of graph planning, graphplan,[2] a Java implementation of it, PDDLGraphPlan,[3] and Blackbox,[4] a planner combining graphplan building and the use of SAT solvers to retrieve plans. The pycompose architecture enables to support other planners through the implementation of a class with two methods: problemToString and run, respectively to output a problem in planner format and to run and parse planner results.

Application. If we run pycompose on our composition problem with the first requirement workflow (Fig. 17.4, left), we get one solution (computed in 0.11 s on a 2.53 GHz Mac Book Pro, including 0.03 s for the planner to retrieve the plan):

$$\begin{aligned}&(\text{pear\_product}{:=}\mathit{cast}(\text{etablet}) \mid\mid \{\text{user\_name},\text{user\_address},\text{credit\_card\_info},\text{pim\_wallet}\}\\ &\qquad{:=}\mathit{dec}(\text{user\_info}))\ ;\\ &(\text{shipping\_addr}{:=}\mathit{cast}(\text{user\_address}) \mid\mid \text{billing\_addr}{:=}\mathit{cast}(\text{user\_address}) \mid\mid w_1{:}\text{order})\ ;\\ &w_1{:}\text{ship}\ ;\ w_1{:}\text{bill}\ ;\ w_1{:}\text{charge}\ ;\ w_1{:}\text{ack}\end{aligned}$$

The workflow representation of this solution is presented in Fig. 17.8.

However, let us now suppose that the user does not want to give his credit card (user_info $\lhd_{cc}$ credit_card_info is removed from DSS, or the user input is replaced with {etablet,user_name,user_address,pim_wallet}). There is no longer any possible composition: w_1 cannot proceed with payment (no credit card information), moreover, w_2 and w_3 cannot interact since this would yield that capability *payment* is

[2] http://www.cs.cmu.edu/avrim/graphplan.html

[3] http://www.cs.bham.ac.uk/zas/software/graphplanner.html

[4] http://www.cs.rochester.edu/kautz/satplan/blackbox/

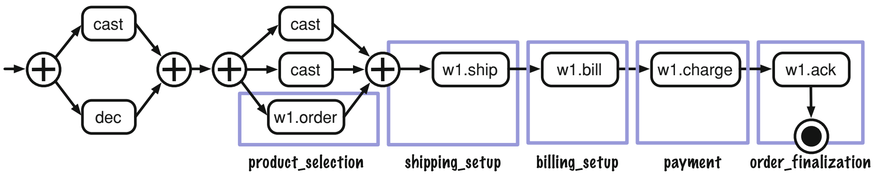

Fig. 17.8 eTablet buying—composition solution

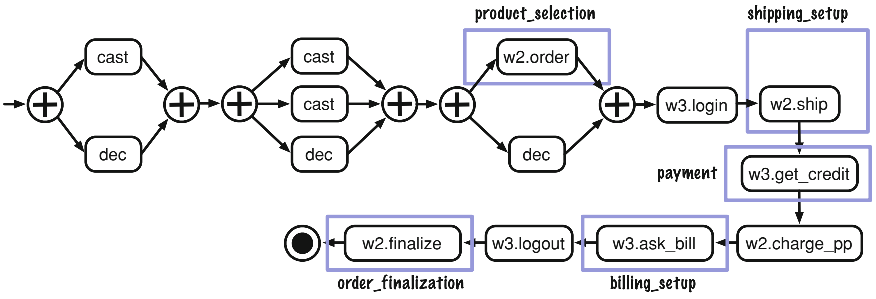

Fig. 17.9 eTablet buying—alternative composition solution

done before capability *billing_setup* (see w_3 workflow in Fig. 17.3 and its operations in Table 17.3) while the requirement workflow forbids it. However, if we let a more permissive requirement workflow (Fig. 17.4, right) then we get a composition (computed in 0.11 s on a 2.53 GHz Mac Book Pro, including 0.04 s for the planner to retrieve the plan) where w_2 and w_3 interact:

(pear_product := *cast*(etablet) || {user_name,user_address,credit_card_info,pim_wallet}
:= *dec*(user_info)) ;
(product := *cast*(pear_product) || shipping_addr := *cast*(user_address)
|| {paypal_info,amazon_info} := *dec*(pim_wallet)) ;
(w_2:order || {paypal_login,paypal_pwd} := *dec*(paypal_info)) ;
w_3:login ; w_2:ship ; w_3:get_credit ; w_2:charge_pp ; w_3:ask_bill ; w_3:logout ; w_2:finalize

The workflow representation of this second solution is given in Fig. 17.9.

17.5 QoS Optimization of Conversational Service Composition as an Extension

In this section we first introduce how to calculate the aggregation of Quality of Services (QoS). Then we extend the developed composition method to include QoS optimization as a non-functional goal.

17.5.1 Aggregation of Quality of Services

A conversational service is composed of a set of operations over which a workflow is specified. Each operation o can be regarded as an elementary service w with certain qualities. For a network of conversational services, we can calculate the QoS of the network as if we have a network of elementary services. Suppose we use $\sigma = w_1, w_2, \ldots, w_n$ to represent a network of connected elementary services. If they are connected in sequence, $\sigma = w_1; w_2; \ldots; w_n$, or in parallel, $\sigma = w_1||w_2||\ldots||w_n$. For an elementary service w, a finite set of quality criteria of w is denoted as $Q(w)$. Since our work focuses on throughput and execution price, the two quality criteria for an elementary service w and the aggregated value over σ:

- **Throughput** $Q_1(w)$: the average rate of successful message delivery over a communication channel, e.g., 10 successful invocations per second.

$$Q_1(w_1; \ldots; w_n) = \min Q_1(w_i) \tag{17.1}$$
$$Q_1(w_1|| \ldots ||w_n) = \min Q_1(w_i) \tag{17.2}$$

- **Execution price** $Q_2(w)$: the fee to invoke w.

$$Q_2(w_1, \ldots, w_n) = \sum Q_2(w_i) \tag{17.3}$$

Some of the above criteria are negative, i.e., the higher the value, the lower the quality. Execution price and response time are in this category. The other criteria, such as throughput, are positive, i.e., the higher the value, the higher the quality. We want to have a uniform way to compare the qualities, especially with the multiple criteria. We apply a Multiple Criteria Decision Making (MCDM) technique [33] to aggregate QoS value $Q(w)$. Similar to [8] and [36], we first scale the value of a quality i for a service w_j. For negative criteria, e.g., execution price, values are scaled according to Eq. 17.4. For positive criteria, e.g., throughput, values are scaled according to Eq. 17.5. For all the criteria, the higher the quality value, the lower the utility value $U_i(w_j)$. This is because the classic Dijkstra's algorithm finds the "shortest distance"(lowest cost) over a graph.

$$U_i(w_j) = \begin{cases} \frac{Q_i(w_j) - Q_i^{min}}{Q_i^{max} - Q_i^{min}} & \text{if } Q_i^{max} - Q_i^{min} \neq 0 \\ 1 & \text{if } Q_i^{max} - Q_i^{min} = 0 \end{cases} \tag{17.4}$$

$$U_i(w_j) = \begin{cases} \frac{Q_i^{max} - Q_i(w_j)}{Q_i^{max} - Q_i^{min}} & \text{if } Q_i^{max} - Q_i^{min} \neq 0 \\ 1 & \text{if } Q_i^{max} - Q_i^{min} = 0 \end{cases} \tag{17.5}$$

The overall quality score for a Web service w_j is defined in Eq. 17.6:

$$U(w_j) = \sum U_i(w_j) \times W_i \tag{17.6}$$

where $W_i \in [0, 1]$ and $\sum W_i = 1$. W_i represents the weight of criterion i.

For a network of services σ, we would like to do the same conversion. The following equations for defining $U_i(\sigma)$ are the same as those defining $Q_i(\sigma)$, just changing $Q_i(w_i)$ to $U_i(w_i)$, except in Eqs. 17.9 and 17.10, max replaces min, it is because U_2 decreases when Q_1 increases. Thus, the max value of U_1 corresponds to the min value of Q_2.

$$U_1(w_1; \ldots; w_n) = \sum U_1(w_i) \tag{17.7}$$

$$U_1(w_1 || \ldots || w_n) = \max U_1(w_i) \tag{17.8}$$

$$U_2(w_1; \ldots; w_n) = \max U_2(w_i) \tag{17.9}$$

$$U_2(w_1 || \ldots || w_n) = \max U_2(w_i) \tag{17.10}$$

The aggregated utility value for σ is:

$$U(w_1, \ldots, w_n) = \sum U_i(w_1, \ldots, w_n) \times W_i \tag{17.11}$$

With Eqs. 17.7–17.11, we could compare two networks of services by the individual utility values U_i or by the overall score U. The lower the value, the higher the quality of service. We uniform the increasing and decreasing sense of the quality criteria. But the calculation of the precise values are still different for different criteria.

17.5.2 Encoding QoS-Aware Composition as a QoS-Aware Planning Problem

When QoS is given, people expect a solution with the best quality. A QoS-aware service composition is to generate a business process that fulfills the functional goals and optimizes the QoS value simultaneously. We first explain the intuition behind the QoS-aware planning technique. Then we present how to solve QoS-aware composition using QoS-aware planning technique.

17.5.2.1 Motivations

When service composition is encoded as a graph planning problem, the planning graph built by the graph planning technique records all the functional elements, i.e., all the execution paths, to achieve functional goals. If the planning graph can be extended into a QoS-aware planning graph that not only records functional elements but also includes the QoS information, it is possible to solve QoS-aware service composition problem using planning graph technique.

We discover that combining Dijkstra's algorithm with the planning graph technique provides a good way to associate the QoS value with each vertex in a planning graph. Firstly, a planning graph is a compact representation of the whole problem space. All the execution paths that achieve goals can be extracted from the planning graph. We could use a systematic search algorithm like Dijkstra's algorithm to traverse the planning graph from the initial layer to the goals. Secondly, the principle of Dijkstra's algorithm is to calculate the best *cost-to-come* value for a vertex. Since a proposition can be regarded as a vertex of the planning graph, we could use the same principle to calculate the best *cost-to-come* value which is the best QoS value for each proposition. Then, we could get the overall *cost-to-come* for all the goal propositions. And during the search, we could record the best path which is the best plan.

17.5.2.2 QoS-Aware Graph Planning Technique

Our QoS-aware Graph Planning technique builds a Tagged Planning Graph (TPG) instead of a normal planning graph. This technique is firstly developed in our paper [34] for QoS-aware composition problems without negative effects. In this paper we extend it to work with negative effects. A TPG is an extension of a planning graph in the sense that each vertex in the planning graph is assigned with a tag. The affiliated tag records the related QoS information for each vertex. In the following, we present the way to calculate the tag values for action vertices and for proposition vertices respectively.

Actions

When service composition is encoded as a Graph Planning problem, actions in the planning graph come from DSS encoding, workflow encoding, service encoding and composition requirement encoding. The tag for each action vertex is the QoS value of the action. Except the actions encoded from the operations of the conversational services, the other actions, such as decomposition, composition and casting from DSS encoding, P_{so}, P_{sa}, P_{jo} and P_{ja} from service encoding, etc., do not have QoS values. We need to assign a default QoS value to an action vertex which does not have a QoS value in order to facilitate QoS aggregation.

For negative criteria, e.g., execution price, the default QoS value is zero. For position criteria, e.g., throughput, the default QoS value is the maximal value of all actions encoded from operations. These assignments make sense, because these values do not affect the calculation of the aggregated QoS of the resulting composite service.

Propositions

The tag T_p for a proposition vertex p is a set $\{t_1^p, \ldots, t_k^p\}$ $(k = ||T_p||)$, which represents all possible execution paths leading to the proposition p. Each tag member t_j^p $(j = 1, \ldots, k)$ in T_p is a tuple $(QoSValue, executionpath)$ and corresponds to an execution path $executionpath$ with its QoS value $QoSValue$. An $executionpath$ is actually a plan $\Pi = \pi_1; \ldots; \pi_n$ to achieve p, where π_j $(j = 1, \ldots, n)$ is a set of actions that can be executed in parallel. When we search a plan for p, we exclude the invalid plans that contain mutex pairs of actions due to the negative effects of actions. Also, we calculate the QoS values for these valid plans at the same time.

For a proposition p at layer P_0, $T_p = \{(U, \{\})\}$ where U is a default QoS value. The assignment of the default QoS value for a proposition at P_0 is similar to the assignment of the default QoS value for actions. For negative criteria, such as response time or execution price, the default QoS value is zero. For position criteria, such as throughput, the default QoS value is the maximal value of all actions encoded from operations. These values do not affect the calculation of the aggregated QoS of the resulting composite service. The $executionpath$ is $\{\}$ since p is provided by service composition query.

Inspired by the Dijkstra's algorithm, we calculate the tag for a p at layer P_i $(i \geq 1)$ when the planning graph is constructed. If an action a at layer A_i $(i \geq 1)$ produces p at layer P_i $(i \geq 1)$, we calculate the tag for p as the following.

- **Calculate the execution paths**. If action a produces p, the combinations of the execution paths of $pre(a)$ (in parallel) appended by a are the execution paths of p. If there are several actions produce p, the execution paths calculated from these actions are all execution paths for p. If these actions are mutex, the execution paths are mutex too.
- **Calculate the QoS value for each execution path**. The calculation of the QoS value for each execution path follows the QoS aggregation formulas. One execution path leading to p is consist of one combination of the execution paths of $pre(a)$ and a. For example, if throughput is the QoS criterion, the throughput of an execution path leading to p is the minimum of the throughput of the combined execution paths and the throughput of a. If execution price is the QoS criterion, that the total execution price of all the combined execution paths plus the execution price of a is the execution price of the execution path leading to p.

After the TPG is constructed, we extract an optimal plan by backtracking the execution path for each goal proposition. An optimal plan is consist of the optimal plans to achieve each individually goal simultaneously. Since a plan cannot contain any mutex pair of actions, we need to consider all the possibilities.

17.5.2.3 QoS-Aware Graph Planning

QoS-aware Graph Planning extends the standard Graph Planning technique. In the construction phase, the QoS tag is calculated and the graph is constructed until a fixedpoint layer, because a longer plan may have a better QoS value. In the backtracking phase, a solution with the best QoS value is extracted. For simplicity, we present our algorithms using throughput as the single quality criterion and the calculation of throughput follows Eqs. 17.1 and 17.2. The calculation of the other QoS criteria is discussed in Sect. 17.5.3.

Algorithm 1 called $QoSGraphPlan$ is the main algorithm QoS-aware Planning Graph. Line 1 sets U as the the maximum throughput of actions. At layer P_0, the multiple tags T_p only contains a tuple of U and an empty set (line 2). Starting from layer 0 (line 3), the algorithm calls $ExpandGraph$ (Algorithm 2) to construct a TPG layer by layer until it reaches $Fixedpoint$ (line 4–7). If the fixed-point layer P_n contains all goal propositions without mutex (line 8), the algorithm calls $ExtractPlan$ (Algorithm 4) (line 9) to extract an optimal plan from the TPG. Otherwise, there is no plan exist (line 11).

Algorithm 1: $QoSGraphPlan(A, s_0, g)$

```
Data: $G = \langle P_0, A_1, \mu A_1, ..., A_n, \mu A_n, P_i, \mu P_n \rangle$ is a planning graph;

1: $U \leftarrow \max\{cost(a) | a \in A\}$;
2: $P_0 \leftarrow \{(p, T_p) | p \in s_0, T_p$ is a multiple-tag set of $p$ where $T_p \leftarrow \{(U, \{\})\}\}$;
3: $i \leftarrow 0$;
4: repeat
5:    $i \leftarrow i + 1$;
6:    $G \leftarrow ExpandGraph(G)$;
7: until $Fixedpoint(G)$
8: if $g \subseteq P_n$ and $g^2 \cap \mu P_n = \emptyset$ then
9:    print $ExtractPlan(G, g)$;
10: else
11:    print $\emptyset$;
12: end if
```

Algorithm 2 called $ExpandGraph$ expands the TPG by one layer. A_i gets all the enabled actions at layer i and each actions has a tag t (line 1). The tag t is the QoS value, i.e., throughput, of the action. The enabled actions are those whose inputs are in the previous layer $i - 1$ and there is no mutual exclusion between propositions belonging to the inputs. μA_i is the set of mutex pairs of actions in A_i (line 2). P_i contains positive effects of actions in A_i (line 3). We assign a tag T_p to each $p \in P_i$. T_p is actually a multiple-tag set. Each element $t_j^p \in T_p$ $(j = 1, \ldots, ||T_p||)$ is a tuple $(t_j^p.v, t_j^p.\Pi)$, where $t_j^p.\Pi$ is a execution path that leads to p and $t_j^p.v$ is the QoS value of $t_j^p.\Pi$. It calls $CalMultiTag$ (Algorithm 3) to calculate T_p for p. Line 4 gets the set of mutex pairs of propositions in P_i, denoted as μP_i. Line 5–line 9 create the arcs between actions and propositions.

Algorithm 2: $ExpandGraph(G)$

```
Data: $G = \langle P_0, A_1, \mu A_1, ..., A_n, \mu A_n, P_i, \mu P_n \rangle$;
1: $A_i \leftarrow \{(a, t) | pre(a) \subseteq P_{i-1}, pre^2(a) \cap \mu P_{i-1}, t = cost(a)\}$;
2: $\mu A_i \leftarrow \{(a, b) \in A_i^2, a \neq b | effects^-(a) \cap [pre(b) \cup effects^+(b)] \neq \emptyset$ or
   $effects^-(b) \cap [pre(a) \cup effects^+(a)] \neq \emptyset$ or $\exists (p, q) \in \mu P_{i-1} : p \in pre(a), q \in$
   $pre(b)\}$;
3: $P_i \leftarrow \{(p, T_p) | \exists a \in A_i : p \in effects^+(a), T_p = CalMultiTag(G, p, i)$ is a multiple-
   tag set of $p$ where $T_p$ is represented as $\{t_j^p | j = 1, \ldots, ||T_p||$ and $t_j^p = (t_j^p.v, t_j^p.\Pi)\}\}$;
4: $\mu P_i \leftarrow \{(p, q) \in P_i^2, p \neq q | \forall a, b \in A, a \neq b : p \in effects^+(a), q \in effects^+(b) \Rightarrow$
   $(a, b) \in \mu A_i\}$;
5: for each $a \in A_i$ do
6:     link $a$ with precondition arcs to $pre(a)$ in $P_{i-1}$;
7:     link $a$ with positive arcs to each of its $effects^+(a)$ in $P_i$;
8:     link $a$ with negative arcs to each of its $effects^-(a)$ in $P_i$;
9: end for
10: return $(\langle P_0, A_1, \mu A_1 ..., A_n, \mu A_n, P_n, \mu P_n \rangle)$;
```

Algorithm 3 calculates the tag value for each proposition p. Initially, T is an empty set (line 1). S is a subset of A_i and each action in S produces p as one of its positive effects (line 2). For each pair $(a, t) \in S$, we get a subset of P_{i-1}, denoted as PT. Each element (p, T_p) in PT satisfies $p \in pre(a)$ (line 4). T_p is a multiple-tag set for p where $T_p = \{t_j^p | t_j^p = (t_j^p.v, t_j^p.\Pi)$ is the j-th element of T_p and $j = \{1, \ldots, ||T_p||\}$. For each element t_j^p of T_p, $t_j^p.\Pi$ is a execution path that leads to p and $t_j^p.v$ is the cost, i.e., throughput, of the execution path $t_j^p.\Pi$. Q is the product of all elements in PT (line 5). For a set of execution paths $\{t_m^{p_1}, \ldots, t_h^{p_k}\} \in Q$ (line 6), a new execution path Π' is obtained by combining all the execution paths (line 7). $CheckMutex(G, \Pi')$ is a function to check whether there exists a mutex pair of actions at the same layer $\Pi'.\pi_j$ $(j = 1, \ldots, ||\Pi'||)$. If $CheckMutex(G, \Pi')$ returns true, it means there exists at least a mutex pair of actions in Π' (line 8). In this case, Π' becomes the execution path Π without mutex pairs of actions (line 9). Accordingly, the cost v (line 10) and the multiple-tag set T (line 11) are updated.

Algorithm 4 extracts a plan with optimal QoS value. First, the optimal plan Π is set to be $\langle\rangle$ (line 1). In line 2, all goal propositions obtained from layer P_n are added into S. Line 3 calculates the direct product of all multiple-tag sets in S. The set of possible plans T is initially an empty set (line 4). For each element $\{t_m^{p_1}, \ldots, t_h^{p_k}\} \in Q$ (line 5), a new possible execution path Π' is obtained by combining all the execution paths (line 6). The cost v is the minimum cost of all sub-execution paths that construct Π' (line 7). Line 8 adds (v', Π') into T. The algorithm starts to find an optimal plan from T until T becomes an empty set (lines 10–20). $SelectOptPlan(T)$ is a function to find the current optimal plan with the maximum throughput from T. For every possible optimal plan with the current maximum throughput returned by $SelectOptPlan(T)$ (line 11), $CheckMutex(G, \Pi)$ is to filter out the plans that contain any mutual exclusion pairs of actions.

Algorithm 3: $CalMultiTag(G, p, i)$

```
Data: $G = \langle P_0, A_1, \mu A_1, ..., A_i, \mu A_i \rangle$;
1: $T \leftarrow \{\}$;
2: $S \leftarrow \{(a,t) | (a,t) \in A_i$ and $p \in effects^{+}(a)\}$;
3: for $(a,t) \in S$ do
4:    $PT \leftarrow \{(p, T_p) | (p, T_p) \in P_{i-1}$ and $p \in pre(a)\}$;
5:    $Q \leftarrow T_{p_1} \times T_{p_2}, \ldots, \times T_{p_k}$ where $(p_j, T_{p_j}) \in PT$ and $j = 1, \ldots, ||PT||$;
6:    for $\{t_m^{p_1}, \ldots, t_h^{p_k}\} \in Q$ do
7:       $\Pi' \leftarrow t_m^{p_1}.\Pi.\pi_1 || \ldots || t_h^{p_k}.\Pi.\pi_1; \ldots; t_m^{p_1}.\Pi.\pi_{i-1} || \ldots || t_h^{p_k}.\Pi.\pi_{i-1}$;
8:       if $CheckMutex(G, \Pi') = true$ then
9:          $\Pi = \Pi'$;
10:         $v \leftarrow \min\{t_j^{p_1}.v | m \le j \le n\}$;
11:         $T \leftarrow T \cup \{(\min\{v, t\}, \Pi; a)\}$;
12:      end if
13:   end for
14: end for
15: return $T$;
```

Algorithm 4: $ExtractPlan(G, g)$

```
Data: $G = \langle P_0, A_1, \mu A_1, ..., A_n, \mu A_n, P_n, \mu P_n \rangle$ is a planning graph.

1: $\Pi \leftarrow \langle\rangle$;
2: $S \leftarrow \{(p, T_p) | (p, T_p) \in P_n$ and $p \in g\}$;
3: $Q \leftarrow T_{p_1} \times T_{p_2}, \ldots, \times T_{p_k}$ where $(p_j, T_{p_j}) \in S$ and $j = 1, \ldots, ||S||$;
4: $T = \{\}$;
5: for $\{t_m^{p_1}, \ldots, t_h^{p_k}\} \in Q$ do
6:    $\Pi' \leftarrow t_m^{p_1}.\Pi.\pi_1 || \ldots || t_h^{p_k}.\Pi.\pi_1; \ldots; t_m^{p_1}.\Pi.\pi_{i-1} || \ldots || t_h^{p_k}.\Pi.\pi_{i-1}$;
7:    $v' = \min\{t_m^{p_1}.v, \ldots, t_h^{p_k}.v\}$;
8:    $T \leftarrow T \cup \{(v', \Pi')\}$;
9: end for
10: repeat
11:   $t \leftarrow SelectOptPlan(T)$;
12:   $\Pi \leftarrow t.\Pi$;
13:   $v \leftarrow t.v$;
14:   if $CheckMutex(G, \Pi) = true$ then
15:      return $(\Pi)$;
16:   else
17:      $\Pi \leftarrow \langle\rangle$;
18:   end if
19:   $T \leftarrow T - \{(v, \Pi)\}$
20: until $||T|| = 0$
21: return $\Pi$;
```

Application. We have a new service Dangdang (w_4, online store). Suppose Dangdang shares the same workflow and operations with ebay service. The operations "order", "ship", "charge_pp", "charge_cc", "bill", and "finalize"of w_4 have the throughput of 4, 6, 26, 13, 10, and 5 repectively. The requirement in our case

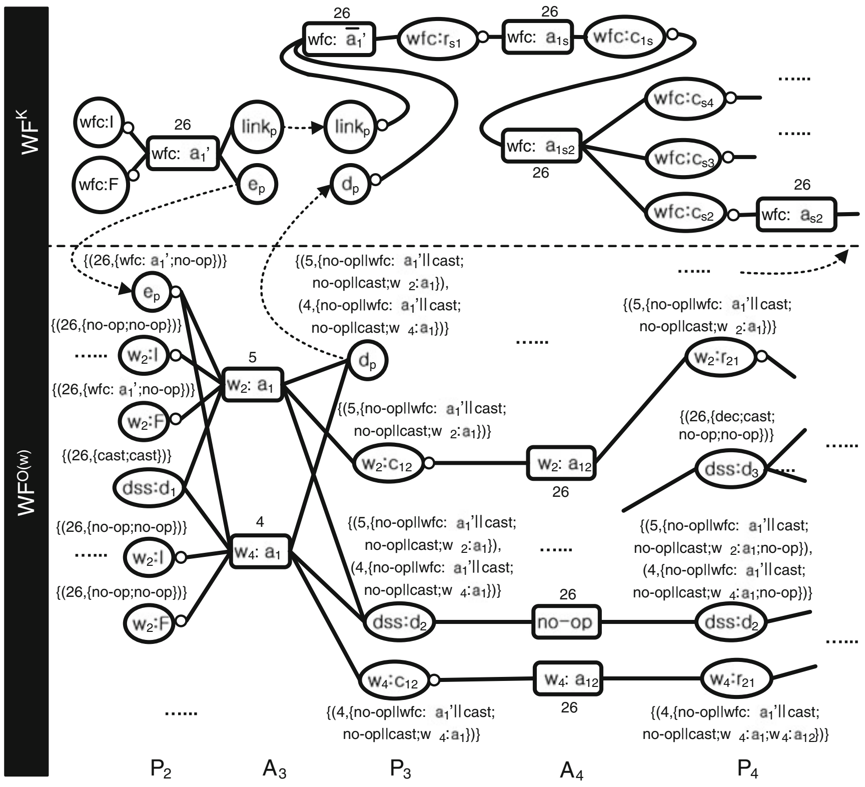

Fig. 17.10 Part of the tagged planning graph

study are $(\{etablet, user\_info\}, \{tracking\_num\}, wfc,)$ and optimization of the throughput of the plan. wfc requires that the payment to be done in parallel to shipping and billing setup as shown in Fig. 17.4 (right). Figure 17.10 presents how to calculate the tag values for each action node in the planning graph. Because the maximum throughput of operations is 26, the tag values of no-op actions and actions that are not encoded from operations are set to be 26. Due to the page limit, we only list part of actions in the planing graph as shown in Table 17.4. Because $effects^-(w_2 : a_1) \cap [pre(w_4 : a_1) \cup effects^+(w_4 : a_1)] = \{e_p\} \neq \emptyset$. The pair $(w_2 : a_1, w_4 : a_1)$ is an element of μA_3. For the proposition d_p in P_3, there are two execution paths without mutex pairs of actions that produce d_p as one of their positive effects. Finally, we remove useless actions from the plan obtained by $ExtractPlan$ algorithm such that the plan only contains operation-derived actions. The workflow of the optimal plan is similar to the one in Fig. 17.9 by replacing w_2 with w_4. The throughput of the optimal plan is 4.

Table 17.4 Part of actions encoded from w_2, w_4 and wfc

	Action	Notation	Cost	$pre(a)$	$effect^{-}(a)$	$effect^{+}(a)$
	order	$w_2{:}a_1$	5	$w_2{:}I$, $w_2{:}F$, e_p, product ($dss{:}d_1$)	$w_2{:}I$, $w_2{:}F$, e_p	e_sessionid($dss{:}d_2$), d_p,$w_2{:}c_{12}$
w_2	order→ship	$w_2{:}a_{12}$	26	$w_2{:}c_{12}$	$w_2{:}c_{12}$	$w_2{:}r_{21}$
	ship	$w_2{:}a_2$	7	shipping_addr($dss{:}d_3$), e_sessionid($dss{:}d_2$), e_s, $w_2{:}r_{21}$	e_s, $w_2{:}r_{21}$	d_s, $w_2{:}c_{2s}$, order_amount($dss{:}d_4$)
	order	$w_4{:}a_1$	4	$w_4{:}I$, $w_4{:}F$,e_p, d_p, $w_4{:}c_{12}$	$w_4{:}I$, $w_4{:}F$, e_p	e_sessionid($dss{:}d_2$), d_p, $w_4{:}c_{12}$
w_4	order→ship	$w_4{:}a_{12}$	26	$w_4{:}c_{12}$	$w_4{:}c_{12}$	$w_4{:}r_{21}$
	ship	$w_4{:}a_2$	6	shipping_addr($dss{:}d_3$), e_sessionid($dss{:}d_2$), e_s, $w_4{:}r_{21}$	e_s, $w_4{:}r_{21}$	d_s, $w_4{:}c_{2s}$, order_amount($dss{:}d_4$)
	product_selection	$wfc{:}a'_1$	26	$wfc{:}I$, $wfc{:}F$	$wfc{:}I$, $wfc{:}F$	e_p,$link_1$
		$wfc{:}\bar{a}'_1$	26	$link_1$, d_p	$link_1$, d_p	$wfc{:}c_{1s}$
wfc	product_selection→ ⊕	$wfc{:}a_{1s}$	26	$wfc{:}c_{1s}$	$wfc{:}c_{1s}$	$wfc{:}r_{s1}$
	⊕	$wfc{:}a_{1s2}$	26	$wfc{:}r_{s1}$	$wfc{:}r_{s1}$	$wfc{:}c_{s2}$, $wfc{:}c_{s3}$, $wfc{:}c_{s4}$
	⊕ →shipping_setup	$wfc{:}a_{s2}$	26	$wfc{:}c_{s2}$	$wfc{:}c_{s2}$	$wfc{:}r_{25}$
				…		

17.5.3 Other QoS Criteria

Up to now, we use throughput as the quality criterion to develop our method. We can also consider how to calculate the other criteria.

For **execution price** (Eq. 17.3), min() function is used. The tag values for the actions encoded from operations are their corresponding execution price. The tag values of other actions are set to be 0. Then, the cost v of a execution path Π is the total execution prices of the plan. Finally, the optimal plan is the plan with the minimum overall execution price.

For **successful execution rate** and **availability**, each service contributes the QoS value in the same way, no matter how they are connected (i.e., sequential or parallel). Here, we focus on throughput and execution price as quality criteria.

17.6 Related Work

Our work is at the intersection of two domains: service composition and software adaptation. Automatic composition is an important issue in Service-Oriented Computing and numerous works have addressed this over the last years [17, 22, 30]. Planning-based approaches have particularly been studied due to their support for underspecified requirements [15, 29]. Automatic composition has also been achieved using matching and graph/automata-based algorithms [4, 11, 25] or logic reasoning [5]. Various criteria could be used to differentiate these approaches, yet, due to our Task-Oriented Computing motivation, we will focus on issues related to service and composition requirement models, and to adaptation.

While both data input/output and capability requirements should be supported, as in our approach, to ensure composition is correct wrt. the user needs, only [6, 25] do, while [4, 11, 16, 20, 21, 37] support data only and [5] supports capabilities only. As far as adaptation is concerned, [4, 16, 21, 25] support a form of horizontal (data) adaptation, using semantics associated to data; and [20] a form of vertical (capability abstraction) adaptation, due to its hierarchical planning inheritance. We combined both techniques to achieve both adaptation kinds. Few approaches support expressive models in which protocols can be described over capabilities—either for the composition requirement [5] or for both composition and services [6, 25] like us. [4, 11, 20, 16, 21] only support conversations over operations (for a given capability).

As opposed to the aforementioned works dealing with orchestration, in [24], the authors present a technique with adaptation features for automatic service choreography. It supports a simple form of horizontal adaptation, however their objective is to maximize data exchange between services but they are not able to compose services depending on an abstract user task.

Most software adaptation works, e.g., [10, 14, 32] are pure model-based approaches whose objective is to solve protocol mismatch between a fixed set of components, and that do not tackle service discovery, composition requirements, or service composition implementation. Few works explicitly add adaptation features to Service-Oriented Computing [12, 23, 27]. They adopt a different and complementary view wrt. ours since their objective is not to integrate adaptation within composition in order to increase the orchestration possibilities, but to tackle protocol adaptation between clients and services, e.g., to react to service replacement.

In an earlier work [1] we already used graph planning to perform service composition with both vertical and horizontal adaptation. With reference to this work, we add support for conversations in both service descriptions and composition requirements. Moreover, adaptation was supported in an ad-hoc fashion, yielding complexity issues when backtracking to get composition solutions. Using encodings, we are able in our work to support adaptation with regular graph planning which enables us to use state-of-the-art graph planning tools.

17.7 Conclusion

Software adaptation is a promising approach to augment service interoperability and composition possibilities. In this paper we have proposed a technique to integrate adaptation features in the service composition process. With reference to related work, we support both horizontal (data exchange between services and orchestrator) and vertical adaptation (abstraction level mismatch between user need and service capabilities). This has been achieved combining semantic descriptions (for data and capabilities) and graph planning. We also support conversations in both service descriptions and composition requirements.

The approach at hand is dedicated to deployment time, where services are discovered and then composed out of a set of services that may change. Yet, in a pervasive environment, services may appear and disappear also during composition execution, e.g., due to the user mobility, yielding broken service compositions. We made a first step towards repairing them in [35], still with a simpler service and composition requirement model (no conversations). A first perspective concerns extending this approach to our new model. Further, we plan to study the integration of our composition and repair algorithms as an optional module in existing runtime monitoring and adaptation frameworks for services composition such as [26].

Acknowledgments This work is supported by project "Service Oriented Systems Integration" (RGPIN/298362-2012) of Canada NSERC Discovery Grant, and by project "Personal Information Management through Internet" (ANR-2010-VERS-0014-03, PIMI) of the French National Agency for Research.

References

1. Beauche, S., Poizat, P.: Automated service composition with adaptive planning, pp. 530–537. In: Proceedings of the ICSOC (2008)

2. Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M.: Towards an Engineering Approach to Component Adaptation. In: Architecting Systems with Trustworthy Components, vol. 3939. LNCS (2006)
3. ter Beek, M.H., Bucchiarone, A., Gnesi, S.: Formal methods for service composition. Ann. Math. Comput. Teleinf. **1**(5), 1–10 (2007)
4. Benigni, F., Brogi, A., Corfini, S.: Discovering service compositions that feature a desired behaviour. In: Proceedings of the ICSOC (2007)
5. Berardi, D., Giacomo, G.D., Lenzerini, M., Mecella, M., Calvanese, D.: Synthesis of underspecified composite e-services based on automated reasoning. In: Proceedings of the ICSOC (2004)
6. Bertoli, P., Pistore, M., Traverso, P.: Automated composition of web services via planning in asynchronous domains. Artif. Intell. **174**(3–4), 316–361 (2010)
7. Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. Artif. Intell. J. **90**(1–2), 225–279 (1997)
8. Bouguettaya, A., Yu, Q., Liu, X., Malik, Z.: Service-centric framework for a digital government application. IEEE Trans. Serv. Comput. **4**(1), 3–16 (2011)
9. Bozkurt, M., Harman, M., Hassoun, Y.: Testing web services: a survey. Technical Report TR-10-01, Centre for Research on Evolution, Search & Testing, King's College London (2010)
10. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. J. Syst. Softw. **74**(1), 45–54 (2005)
11. Brogi, A., Popescu, R.: Towards semi-automated workflow-based aggregation of web services. In: Proceedings of the ICSOC (2005)
12. Brogi, A., Popescu, R.: Automated Generation of BPEL Adapters. In: Proceedings of the ICSOC (2006)
13. Canal, C., Murillo, J.M., Poizat, P.: Software adaptation. L'Objet **12**, 9–31 (2006)
14. Canal, C., Poizat, P., Salaün, G.: Model-based adaptation of behavioural mismatching components. IEEE Trans. Softw. Eng. **34**(4), 546–563 (2008)
15. Chan, K.S.M., Bishop, J., Baresi, L.: Survey and comparison of planning techniques for web service composition. Technical report, Dept Computer Science, University of Pretoria (2007)
16. Constantinescu, I., Binder, W., Faltings, B.: Service composition with directories. In: Proceedings of the SC (2006)
17. Dustdar, S., Schreiner, W.: A survey on web services composition. Int. J. Web Grid Serv. **1**(1), 1–30 (2005)
18. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers, Amsterdam (2004)
19. Kiepuszewski, B.: Expressiveness and suitability of languages for control flow modelling in workflow. PhD thesis, Queensland University of Technology, Brisbane, Australia (2003)
20. Klush, M., Gerber, A., Schmidt, M.: Semantic web service composition planning with OWLS-Xplan. In: Proceedings of the AAAI Fall Symposium on Agents and the Semantic Web (2005)
21. Liu, Z., Ranganathan, A., Riabov, A.: Modeling web services using semantic graph transformation to aid automatic composition. In: Proceedings of the ICWS (2007)
22. Marconi, A., Pistore, M.: Synthesis and composition of web services. In: Proceedings of the 9th International School on Formal Methods for the Design of Computer, Communications and Software Systems: Web Services (SFM)
23. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques, pp. 84–99. In: Proceedings of the ICSOC (2008)
24. Melliti, T., Poizat, P., Ben Mokhtar, S.: Distributed behavioural adaptation for the automatic composition of semantic services. In: Proceedings of the FASE (2008)
25. Mokhtar, B.S., Georgantas, N., Issarny, V.: COCOA: conversation-based service composition in pervasive computing environments with QoS support. J. Syst. Softw. **80**(12), 1941–1955 (2007)
26. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for ws-bpel, pp. 815–824. In: Proceedings of the WWW (2008)

27. Nezhad, H.R.M., Xu, G.Y., Benatallah, B.: Protocol-aware matching of web service interfaces for adapter development, pp 731–740. In: Proceedings of the WWW (2010)
28. Papazoglou, M.P., Georgakopoulos, D.: Special issue on service-oriented computing. Commun. ACM **46**(10), 25–28 (2003)
29. Peer, J.: Web service composition as AI lanning—a survey. Technical report, University of St.Gallen (2005)
30. Rao, J., Su, X.: A survey of automated web service composition methods. In: Proceedings of the SWSWPC (2004)
31. Seguel, R., Eshuis, R., Grefen, P.: An Overview on Protocol Adaptors for Service Component Integration. Technical report, Eindhoven University of Technology (2008) BETA Working Paper Series WP 265
32. Tivoli, M., Inverardi, P.: Failure-free coordinators synthesis for component-based architectures. Sci. Comput. Program **71**(3), 181–212 (2008)
33. Triantaphyllou, E.: Multi-Criteria Decision Making: A Comparative Study. Springer, New York (2000)
34. Yan, Y., Chen, M.: Anytime QoS optimization over the planGraph for web service composition. In: Proceedings of the ACM SAC, Italy (2012)
35. Yan, Y., Poizat, P., Zhao, L.: Repairing service compositions in a changing world. In: Proceedings of the SERA (2010)
36. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web services composition, pp. 411–421. In: Proceedings of the WWW (2003)
37. Zheng, X., Yan, Y.: An efficient web service composition algorithm based on planning graph, pp 691–699. In: Proceedings of the ICWS (2008)