Adaptive Composition of Conversational Services through Graph Planning Encoding

Pascal Poizat[1,2] and Yuhong Yan[3]

[1] University of Evry Val d'Essonne, Evry, France
[2] LRI UMR 8623 CNRS, Orsay, France
pascal.poizat@lri.fr
[3] Concordia University, Montreal, Canada
yuhong@encs.concordia.ca

Abstract. Service-Oriented Computing supports description, publication, discovery and composition of services 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 thanks to a tool, pycompose, which can interact with state-of-the-art graph planning tools.

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 [1] (SOC) is a cornerstone towards the realization of this vision, through the abstraction of heterogeneous resources as services and automated composition techniques [2,3,4]. However, services being elements of composition developed by different third-parties, their reuse and assembly naturally raises composition mismatch issues [5,6]. 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 [7]. More recently, adaptation has been applied in SOC to solve mismatch between

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 35–50, 2010.

services and clients (*e.g.*, orchestrators) [8,9,10]. 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 [11,12] 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.

Outline. Preliminaries on planning are given in Section 2. After introducing our formal models in Section 3, Section 4 presents our encoding of service composition into a planning problem, and Section 5 addresses tool support. Related work is discussed in Section 6 and we end with conclusions and perspectives.

2 Preliminaries

In this section we give a short introduction to AI planning [13].

Definition 1. *Given a finite set* $L = \{p_1, \ldots, p_n\}$ *of proposition symbols, a* planning problem *[13] 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 they satisfy $effect^-(a) \cap [pre(b) \cup effect^+(b)] = \emptyset$ 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 [13] 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 [14] is a technique that yields a compact representation of relations between actions and represent the whole problem world. A planning graph G is a directed acyclic leveled graph. 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

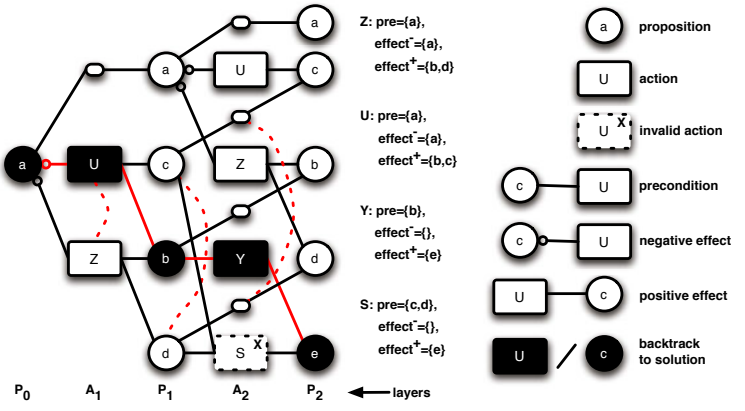

Fig. 1. Graphplan example

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 Figure 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. 1 for U;Y). However, U and Z are in mutual exclusion. Accordingly, since there is no other way to obtain c and d 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 the no-op and U in A_1, or two no-ops in A_2 but we have not represented this for clarity).

3 Modeling

In this section, we present our formal models, grounding service composition. 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.

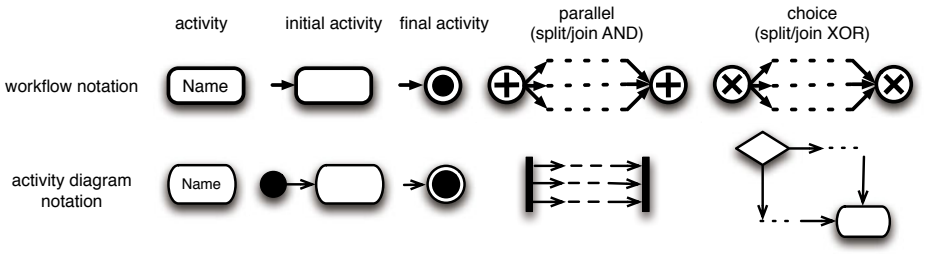

Fig. 2. Workflow notation and relation to the UML activity diagrams

3.1 Conversation Modelling

Different models have been proposed to support service discovery, verification, testing, composition or adaptation in presence of service conversations [15,16,9]. They 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 [17]. 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. 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 [9].

Definition 2. *Given a set of activity names N, a Workflow (WF) [17] 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 [17] 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. 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 $P_A \cup P_{ja} \cup P_{jo}$.

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

Table 1. 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

(CSS). Further, service inputs and outputs are annotated using a Data Semantic Structure (DSS).

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 in 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 (Tab. 1) since concepts can be inferred from these and from the service operation signatures, below.

3.3 Services

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

Definition 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) with $in \subseteq \mathcal{D}$, $out \subseteq \mathcal{D}$, $k \in \mathcal{K}$, 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),

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

Table 2. eTablet buying – services' operations

service	operation	profile
w_1	order	pear_product → pear_product_info, as_sessionid :: product_selection
w_1	cancel	as_sessionid → ∅ :: nil
w_1	ship	shipping_addr, as_sessionid → ∅ :: shipping_setup
w_1	bill	billing_addr, as_sessionid → ∅ :: billing_setup
w_1	charge	credit_card_info, as_sessionid → ∅ :: payment
w_1	gift_wrapper	giftcode, as_sessionid → ∅ :: payment
w_1	ack	as_sessionid → tracking_num :: order_finalization
w_2	order	product → e_sessionid :: product_selection
w_2	ship	shipping_addr, e_sessionid → order_amount :: shipping_setup
w_2	charge_pp	paypal_trans_id, e_sessionid → ∅ :: nil
w_2	charge_cc	credit_card_info, e_sessionid → ∅ :: payment
w_2	bill	billing_addr, e_sessionid → ∅ :: billing_setup
w_2	finalize	e_sessionid → tracking_num :: order_finalization
w_3	login	paypal_login, paypal_pwd → p_sessionid :: nil
w_3	get_credit	order_amount, p_sessionid → paypal_trans_id :: payment
w_3	ask_bill	address, p_sessionid → ∅ :: billing_setup
w_3	logout	p_sessionid → ∅ :: nil

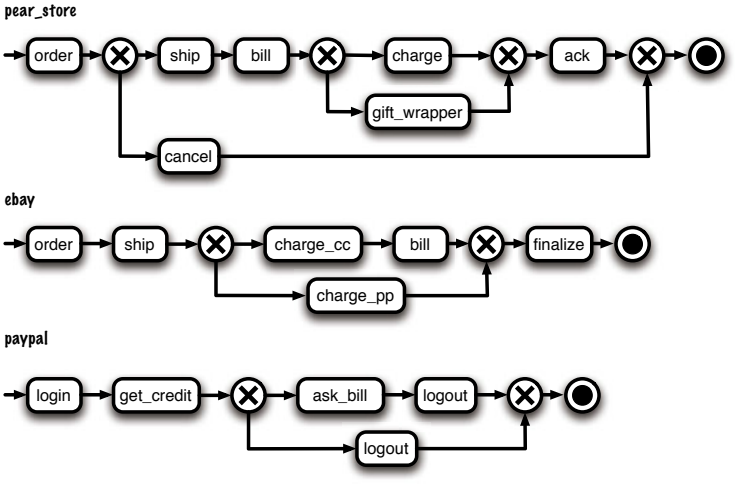

Fig. 3. eTablet buying – services' workflows

$P_{so} = \{\otimes\}$, $P_{jo} = \{\overline{\otimes}\}$, $P_{sa} = P_{ja} = \emptyset$, 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 (we then let $k = \mathsf{nil}$). $o = (in, out, k)$ is also noted $o : in \rightarrow out :: k$.

Application. To fulfil 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 2 and their workflows are given in Figure 3.

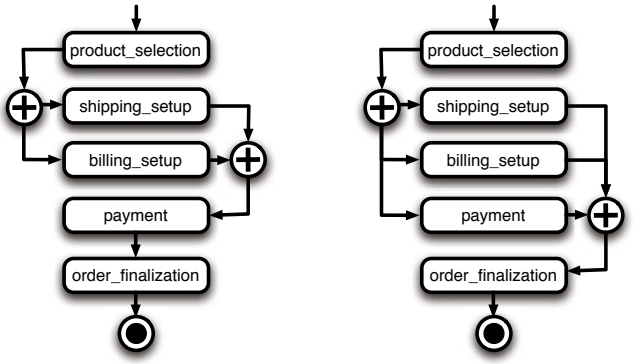

Fig. 4. eTablet buying – requirement workflows

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 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. 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. 4, right) is less strict and enables the payment to be done in parallel to shipping and billing setup.

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

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'.

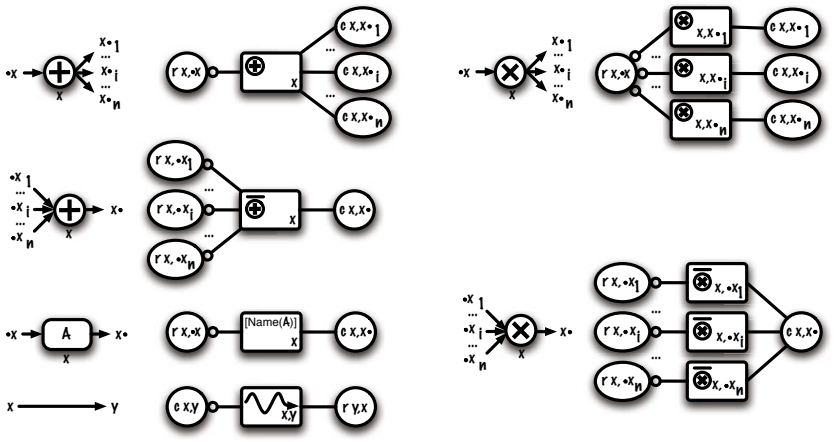

Fig. 5. Workflow encoding

4.2 Workflow Encoding

We reuse here a transformation from workflows to Petri net defined in [17]. 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 behavioural 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. 5):

- for each $x \in P_{sa}$, we have an action $a = \oplus x$, for each $x \in P_{ja}$, we have an action $a = \bar{\oplus} x$, and for each $x \in P_A$, we have an action $a = [Name(x)]x$. 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 $a = \otimes x, y$ and we set $pre(a) = effect^-(a) = \bigcup_{z \in \bullet x}\{r_{x,z}\}$, and $effect^+(a) = \{c_{x,y}\}$.
- for each $x \in P_{jo}$, for each $y \in \bullet x$, we have an action $a = \bar{\otimes} x, y$, 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 $a = \rightsquigarrow x, y$ 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 $effect^+(a)$.

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 4.2). Then

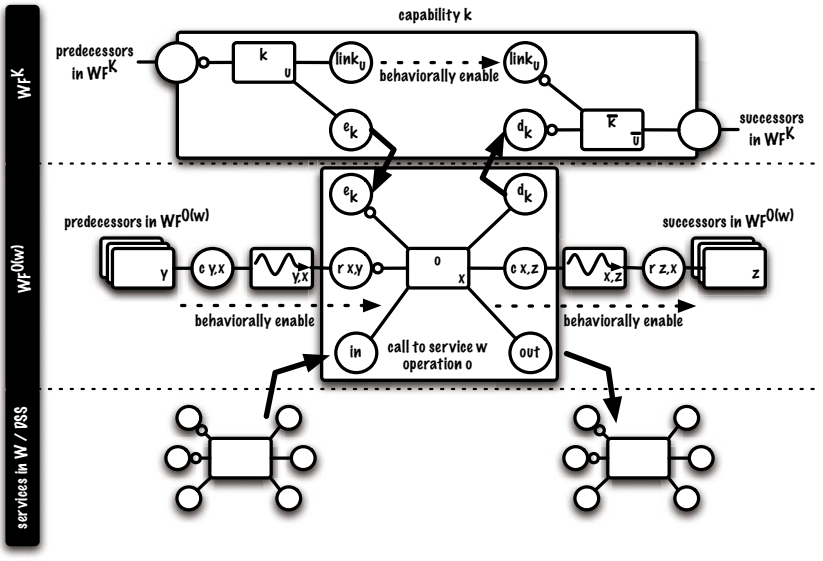

Fig. 6. Principle of interaction between service and requirement encodings

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:

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

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 Figure 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.

4.4 Service Encoding

Each service $w = (O, WF^O)$ is encoded as follows. First we encode the workflow WF^O as presented in 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 4.3 and in Figure 6.

4.5 Overall Encoding

Given a DSS $\mathcal{D}$, a set of services W, and a composition requirement (D_{in}, D_{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 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.

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. 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 [9]. 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 [9] state machine encoding by a more efficient version using sequence and flows. For π we get:

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

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

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

where $modeltrans$ is the transformation of basic assignment / communication activities defined in [9].

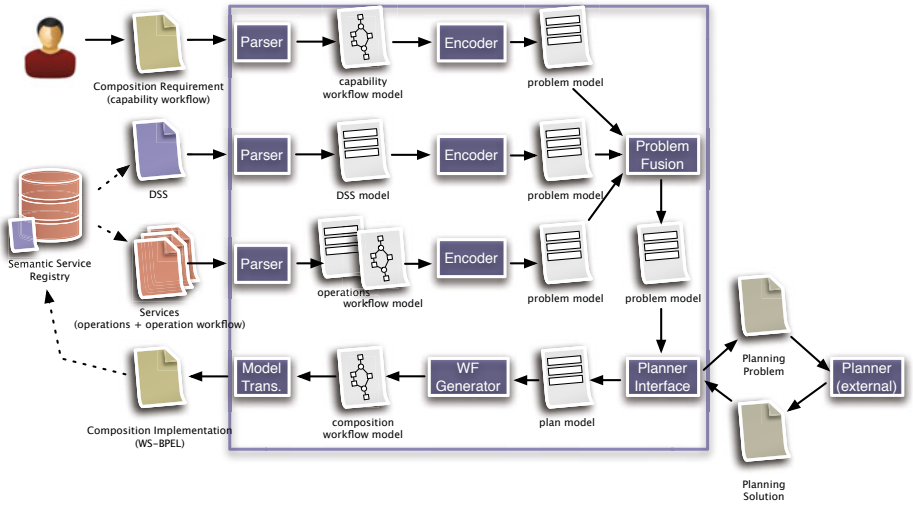

Fig. 7. Architecture of the pycompose tool

5 Tool Support

Our composition approach is supported with a tool, pycompose (Fig. 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 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. 4, left), we get one solution (computed in 0.11s on a 2.53 GHz Mac Book Pro, including 0.03s for the planner to retrieve the plan):

(pear_product:=*cast*(etablet) || {user_name,user_address,credit_card_info,pim_wallet}
:=*dec*(user_info)) ;
(shipping_addr:=*cast*(user_address) || billing_addr:=*cast*(user_address) || w_1:order) ;
w_1:ship ; w_1:bill ; w_1:charge ; w_1:ack

The workflow representation of this solution is presented in Figure 8.

[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/

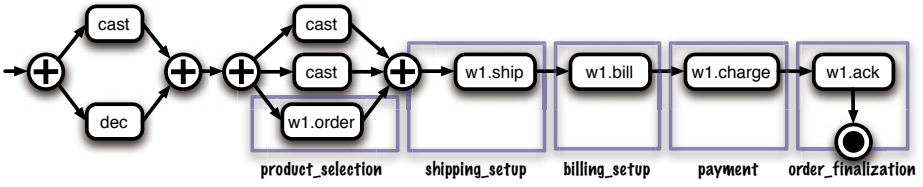

Fig. 8. eTablet buying – composition solution

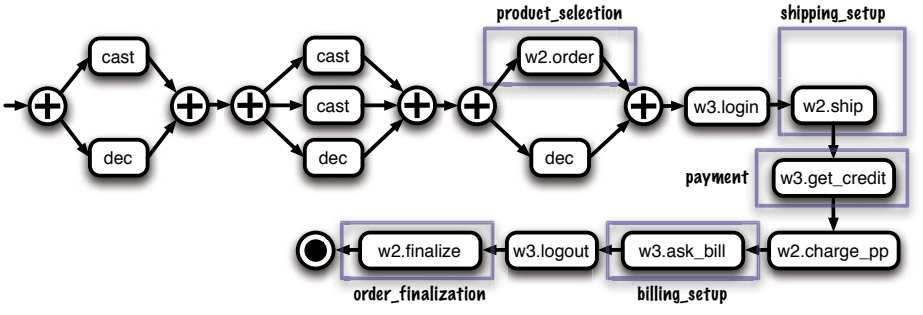

Fig. 9. eTablet buying – alternative composition solution

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 done before capability *billing_setup* (see w_3 workflow in Fig. 3 and its operations in Tab. 2) while the requirement workflow forbids it. However, if we let a more permissive requirement workflow (Fig. 4, right) then we get a composition (computed in 0.11s on a 2.53 GHz Mac Book Pro, including 0.04s 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 Figure 9.

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 [2,3,4].

Planning-based approaches have particularly been studied due to their support for underspecified requirements [11,12]. Automatic composition has also been achieved using matching and graph/automata-based algorithms [18,19,20] or logic reasoning [21]. 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 [22,19] do, while [23,24,25,18,20,26] support data only and [21] supports capabilities only. As far as adaptation is concerned, [24,25,19,20] support a form of horizontal (data) adaptation, using semantics associated to data; and [23] 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 [21] or for both composition and services [22,19] like us. [23,24,25,18,20] only support conversations over operations (for a given capability).

As opposed to the aforementioned works dealing with orchestration, in [27], 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.*, [28,29,30] 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 [8,9,10]. 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 [31] 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.

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 [32], 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 [33].

Acknowledgement. This work is supported by project "Building Self-Manageable Web Service Process" (`RGPIN/298362-2007`) of Canada NSERC Discovery Grant, and by project "PERvasive Service cOmposition" (`ANR-07-JCJC-0155-01`, PERSO) of the French National Agency for Research.

References

1. Papazoglou, M.P., Georgakopoulos, D.: Special Issue on Service-Oriented Computing. Communications of the ACM 46(10) (2003)
2. Rao, J., Su, X.: A survey of automated web service composition methods. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 43–54. Springer, Heidelberg (2005)
3. Dustdar, S., Schreiner, W.: A survey on web services composition. Int. J. Web and Grid Services 1(1), 1–30 (2005)
4. Marconi, A., Pistore, M.: Synthesis and Composition of Web Services. In: Proc. of the 9th International School on Formal Methods for the Design of Computer, Communications and Software Systems: Web Services (SFM)
5. Canal, C., Murillo, J.M., Poizat, P.: Software Adaptation. L'Objet 12, 9–31 (2006)
6. Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M.: Towards an Engineering Approach to Component Adaptation. In: Reussner, R., Stafford, J.A., Szyperski, C. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 193–215. Springer, Heidelberg (2006)
7. 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
8. Brogi, A., Popescu, R.: Automated Generation of BPEL Adapters. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 27–39. Springer, Heidelberg (2006)
9. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 84–99. Springer, Heidelberg (2008)

10. Nezhad, H.R.M., Xu, G.Y., Benatallah, B.: Protocol-aware matching of web service interfaces for adapter development. In: Proc. of WWW, pp. 731–740 (2010)
11. Peer, J.: Web Service Composition as AI Planning – a Survey. Technical report, University of St.Gallen (2005)
12. 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)
13. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers, San Francisco (2004)
14. Blum, A.L., Furst, M.L.: Fast Planning through Planning Graph Analysis. Artificial Intelligence Journal 90(1-2), 225–279 (1997)
15. ter Beek, M.H., Bucchiarone, A., Gnesi, S.: Formal Methods for Service Composition. Annals of Mathematics, Computing & Teleinformatics 1(5), 1–10 (2007)
16. 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)
17. Kiepuszewski, B.: Expressiveness and Suitability of Languages for Control Flow Modelling in Workflow. PhD thesis, Queensland University of Technology, Brisbane, Australia (2003)
18. Brogi, A., Popescu, R.: Towards Semi-automated Workflow-based Aggregation of Web Services. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 214–227. Springer, Heidelberg (2005)
19. Ben Mokhtar, S., Georgantas, N., Issarny, V.: COCOA: COnversation-based Service Composition in PervAsive Computing Environments with QoS Support. Journal of Systems and Software 80(12), 1941–1955 (2007)
20. Benigni, F., Brogi, A., Corfini, S.: Discovering Service Compositions that Feature a Desired Behaviour. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 56–68. Springer, Heidelberg (2007)
21. Berardi, D., Giacomo, G.D., Lenzerini, M., Mecella, M., Calvanese, D.: Synthesis of Underspecified Composite e-Services based on Automated Reasoning. In: Proc. of ICSOC (2004)
22. Bertoli, P., Pistore, M., Traverso, P.: Automated composition of web services via planning in asynchronous domains. Artif. Intell. 174(3-4), 316–361 (2010)
23. Klush, M., Gerber, A., Schmidt, M.: Semantic Web Service Composition Planning with OWLS-Xplan. In: Proc. of the AAAI Fall Symposium on Agents and the Semantic Web (2005)
24. Constantinescu, I., Binder, W., Faltings, B.: Service Composition with Directories. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 163–177. Springer, Heidelberg (2006)
25. Liu, Z., Ranganathan, A., Riabov, A.: Modeling Web Services using Semantic Graph Transformation to Aid Automatic Composition. In: Proc. of ICWS. (2007)
26. Zheng, X., Yan, Y.: An Efficient Web Service Composition Algorithm Based on Planning Graph. In: Proc. of ICWS, pp. 691–699 (2008)
27. Melliti, T., Poizat, P., Ben Mokhtar, S.: Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 146–162. Springer, Heidelberg (2008)
28. Bracciali, A., Brogi, A., Canal, C.: A Formal Approach to Component Adaptation. Journal of Systems and Software 74(1), 45–54 (2005)

29. Canal, C., Poizat, P., Salaün, G.: Model-based Adaptation of Behavioural Mismatching Components. IEEE Transactions on Software Engineering 34(4), 546–563 (2008)
30. Tivoli, M., Inverardi, P.: Failure-free coordinators synthesis for component-based architectures. Science of Computer Programming 71(3), 181–212 (2008)
31. Beauche, S., Poizat, P.: Automated Service Composition with Adaptive Planning. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 530–537. Springer, Heidelberg (2008)
32. Yan, Y., Poizat, P., Zhao, L.: Repairing service compositions in a changing world. In: Proc. of SERA (2010)
33. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for ws-bpel. In: Proc. of WWW, pp. 815–824 (2008)