Automated Service Composition with Adaptive Planning (Long Version*, version 1, June, 16th, 2008)**

Sandrine Beauche[1] and Pascal Poizat[1,2]

[1] INRIA/ARLES project-team, France
{sandrine.beauche,pascal.poizat}@inria.fr
[2] IBISC FRE 3910 CNRS – Université d'Évry Val d'Essonne, France

Abstract. Task-Oriented Computing supports the realization of user needs through the automatic composition of services from service descriptions and user tasks, i.e., high-level descriptions of the user needs. Service-Oriented Computing supports the description, publication, discovery and composition of services. Yet, automatic service composition processes commonly assume that service descriptions and user tasks 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 descriptions and adaptive extensions to graph planning. Our adaptive composition technique is fully automatic and has been implemented in a prototype tool.
keywords: Services, Task-Oriented Computing, Composition, Software Adaptation, Planning, Workflow Languages, 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 existing automated composition techniques. Yet, services being elements of composition developed by different third-parties, their reuse and assembly naturally raises composition mismatch issues [2, 3]. 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.

* A shorter version of this paper has been published in the proceedings of ICSOC'08.
** This work is supported by the project "PERvasive Service cOmposition" (PERSO) of the French National Agency for Research, ANR-07-JCJC-0155-01.

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 [2, 3]. More recently, adaptation has been applied in SOC to solve mismatch between services and clients (*e.g.*, orchestrators) [4, 5]. 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 [6] as it supports the automatic composition of services from underspecified requirements.

The rest of this paper is structured as follows. We present with more details the motivations and the principles of our approach, together with a running example, in Section 1. After giving the necessary preliminaries on planning in Section 2, we present more technically our adaptive planning composition in Section 3. Related work is discussed in Section 4 and we end with conclusions and perspectives.

2 Motivating Example and Overview of the Approach

In this article we use a running example inspired from the case study introduced in [7]. This example is relative to the purchase and online payment of articles from an electronic store, and is representative of user tasks to be composed out of services available over the Internet. Due to lack of place, we focus here on the model-based part of our approach, *i.e.*, we work with models for services and user tasks. More precisely, as far as service conversations and resulting orchestrators are concerned, we work with YAWL [8] models. We think this is a sensible choice due to the the existence of a YAWL extension (simply refereed as YAWL in the sequel) for Web services and a tool supporting translation from/to WS-BPEL [4]. In this paper we are interested in the basic constructs needed to represent external behavioural service descriptions: receive, reply, invoke, sequence, choice (if) and loop (while). The relation between WS-BPEL and YAWL is illustrated in Figures 1 and 2. Data used by an activity are given above them. Accordingly, resulting data is below the actions. The Amazon service provides a capability called BookSearch with a conversation (sequence) over three operations: login and logout, with a customer identifier (customerId) as input (in message part UId), and itemSearch with a book title (title) as input (in message part bookTitle) and a structured information on the search result (bodySearch) as output (in message part result).

Conversations constitute the external view of services, *i.e.*, they describe *how* to use services and hide their internal details (a service may be an orchestrator and itself may call other services' operations). To support the automatic reasoning on services, which includes discovery and composition, additional information is required. The usual approach in automatic service composition is to rely on semantic annotations. Here, we will use them for (i) service capabilities and (ii) the input and output data associated to capabilities.

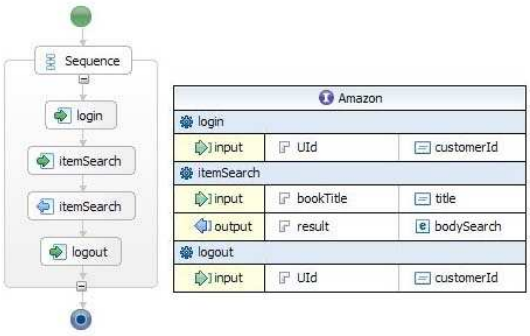

Fig. 1. Amazon service conversation (WS-BPEL)

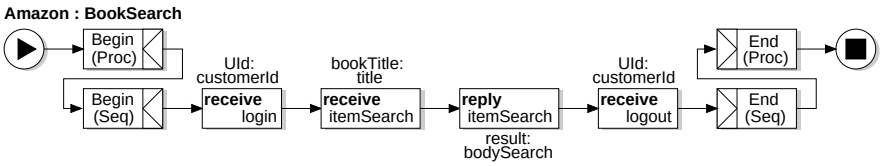

Fig. 2. Amazon service conversation (YAWL)

Let us now end the presentation of our example services (Fig. 3, where we focus on the important part of conversations only). The Amazon service presented above can be used to look for an eBook. There are two possible services for payment (capability OnlinePayment), Paypal and MPS, and service MobiPocket can be used to download an eBook (capability eBookDownload) in PRC format. Using these services, to buy and read an eBook, one may then compose Amazon (to search it), Paypal or MPS (to pay for it), and MobiPocket (to obtain it). Another solution would be to use service @eBook to search and pay at once, and then use the token it generates to get the eBook with MobiPocket.

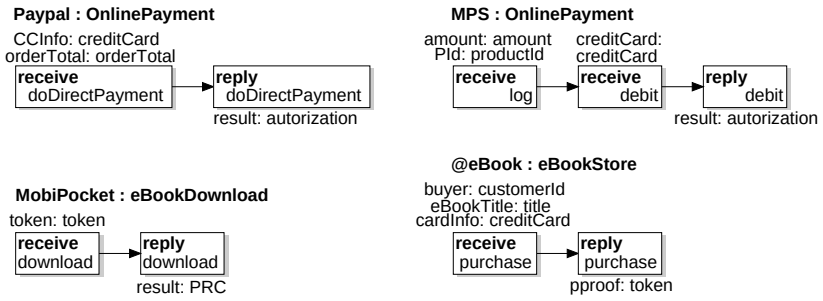

Fig. 3. Service conversations (YAWL)

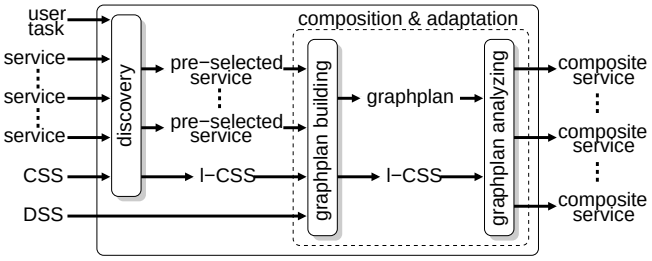

Fig. 4. Overview of our approach

Yet, the user does not know all this rather low-level information, nor the effective service capabilities, neither the exact data that can be exchanged through the orchestrator between composed services to achieve the correct service composition. The user only has a high-level vision of her(his) needs: a capability, the inputs (s)he is ready to provide and the outputs (s)he wants. In this example, (s)he wants an eBookRetrieve capability, to provide title, customerId, and creditCard information, and finally get an eBook in PRC format. There is clearly a (vertical) mismatch between the user's needs and the service descriptions.

Additionally, the services have been developed by different third-parties. One may expect to compose them while from the input/output perspective they could not be chained as-is by the orchestrator. For example, Amazon should be composed with Paypal or MPS but part of the input data they require (respectively orderTotal and amount+productId) does not correspond to what one gets from a call to Amazon (bodySearch). This illustrates a (horizontal) mismatch. Yet, the required data could be inferred from bodySearch.

In order to solve mismatch, we propose to use *software adaptation* techniques. They rely on *adaptation mappings* to solve mismatch out (*e.g*, correspondences between operation names when solving a signature mismatch between a client requiring an operation and a service providing an operation with a different name). More precisely:

- *horizontal adaptation* could be supported through relations between concepts in an ontology of data types (Data Semantic Structure, DSS). These relations would explicit possible transformation rules between data;
- *vertical adaptation* could be supported through a hierarchical structure describing relations between capabilities (Capability Semantic Structure, CSS).

Given a user task, a set of service models (YAWL), and both a DSS and a CSS, our approach (Fig. 4) proceeds as follows. The CSS is first used to filter services that may potentially be used in the composition, based on their capabilities. Additionally, the CSS is labelled with potential (discovered) services for each capability (l-CSS). Then, a graph planning structure is computed, which chains all services capabilities based on input/output dependencies and l-CSS constraints. Finally this structure is analysed to obtain all possible service compositions (in YAWL) corresponding to the user task (which can be none).

3 Preliminaries

Planning is a technique to generate an ordering of a subset of available *tasks*, called a *plan*, in order to reach objectives with respect to some constraints. There are different kinds of planning. In *chaining*, *e.g.*, [9], there is a given world whose state is modified by tasks effects (also called postconditions). The application of tasks can be constrained by their preconditions. In forward chaining, plans are built by chaining tasks, from an initial world state, up to a final one (the objective). Accordingly, backward chaining proceeds from the objective back to the initial state. In *hierarchical planning*, *e.g*, [10, 11], the objective is different as it is the decomposition of abstract tasks into concrete ones following constraints specified in a hierarchical task structure.

Both kinds of planning are important for our objective. Chaining could be used to chain calls to service capabilities from the set of data the user is ready to provide, up to the set of data (s)he wants. For this, task preconditions would correspond to the input data required by service capabilities and task effects to the output data they produce. Yet, this could yield inconsistent chains of service calls, or more generally compositions not correct wrt. the user need specified as an abstract capability. Hierarchical planning is a complementary solution that could orient the chaining towards semantically correct compositions. In the sequel we present representative instances of both techniques.

3.1 Graphplan [9]

Graphplan is a chaining technique that builds a (forward) chaining graph (called *graphplan*) that contains all possible plans and then extracts plans from it. This graph is made up of alternating fact (data) and action layers. Layers themselves contain fact nodes (resp. action nodes). The graph is built by adding in an action layer all actions whose preconditions are present in the current fact layer and accordingly producing in the next fact layer the data corresponding to these actions' effect. Specific actions (no-op) are used to keep data from one layer to the next one, and arcs to relate actions with used data and produced effects. The construction stops when the objective is included in the fact layer (or with a fixpoint). An example is given in Figure 5 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. 5 for U;Y). A major benefit of Graphplan is to represent in a single structure all possible solutions (plans). Graphplan introduces the concept of *mutual exclusion* constraints between nodes. They are reported from a layer to the next one while building the graph. For example, U and Z could be in mutual exclusion (let us suppose the user wants *either* one *or* the other to fulfil its need). 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

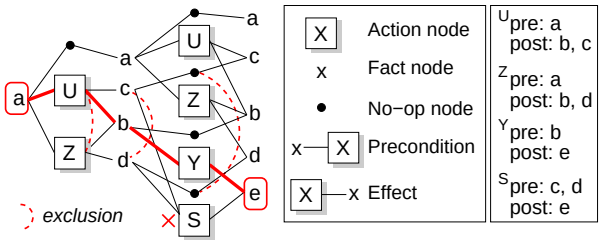

Fig. 5. Graphplan example

S impossible. Moreover, two actions that require facts in mutual exclusion (here two no-ops) are exclusive too.

3.2 GraphHTN [11]

Let us suppose U and Z can be respectively used to buy a book and a CD. While sharing the same inputs, their semantics is different and a user may accept the U;Y plan and not the Z;Y one. Chaining constraints are not enough to support this. Rather we would like to support (abstract) user requirements over the acceptable plan(s). GraphHTN is a hierarchical task planning approach that extends Graphplan with decomposition constraints specified with an AND/OR tree (Fig. 6). Non-leaf nodes correspond to abstract tasks. OR nodes describe potential decompositions of abstract tasks while AND nodes impose that their children are present in the solution. In such a case, a total ordering can be specified (arrow) or not. In Figure 6, W can be decomposed either as U;Y;T or as Z;X.

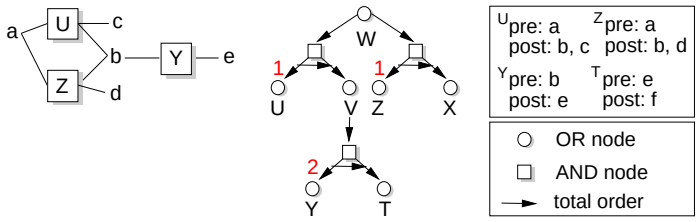

Fig. 6. GraphHTN example – after step 2

The marking of the tree while building plans is used to enforce the tree constraints. Let us look at Figure 6. U and Z are added in the first step because (i) their preconditions are satisfied (a) and (ii) they respect the tree constraints (the first task for W can be U or Z). Accordingly their effect in the graphplan is computed and their nodes are marked with the current step (1). At the next step, V can be applied because its predecessor sibling is marked. Hence Y can be

applied too, since V is abstract. The corresponding node (Y) is then marked (2). The right W branch is discarded since there is no task X. T may now be applied since its predecessor (Y) is marked. This would end the graphplan building as (i) V would be marked (being an AND node with all children marked) and then (ii) W, the root, would be too (being an OR node with one marked child). Stopping as soon as the root is marked, GraphHTN extracts a unique plan. For this, it relies on the same backward analysis as Graphplan.

4 Adaptive Planning Composition

4.1 Models

As discussed in Section 2 our approach takes as input a user task and a set of services to be selected and composed in an orchestration. Moreover, in order to perform adaptation at the same time than composition (the orchestrator is also an adaptor), we rely on a semantic structure for capacities and a semantic structure for data.

We define a *Data Semantic Structure (DSS)* as a tuple $(\mathcal{D}, \lhd, \sqsubset, \rightsquigarrow)$ where $\mathcal{D}$ is a set of concepts (or semantic data type[3]) 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 simulation 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 (d into D if $D = \{d_i \mid d \lhd d_i\}$), composition ($D$ into d if $D = \{d_i \mid d \lhd d_i\}$) and casting ($d_1$ into d_2 if $d_1 \sqsubset d_2$) of data types exchanged between services and orchestrator. Transformation functions, $\rightsquigarrow$, are used to transform some data into another one using relations between XPath terms (*e.g.*, in Fig. 7(a), a ColouredPoint can be transformed into a Pixel). Such transformations are not required in our example (Fig. 7(b)), where, *e.g.*, a price can be used as an orderTotal, but both are basic types. An example of composite type is bodySearch, made up of a price and a productId. DSS are rich structures that support data adaptation and may result from ontology matching [12].

We also define a *Capacity Semantic Structure (CSS)* (Fig. 7(c)) as a couple $(\mathcal{K}, \mathcal{T}_\mathcal{K})$ where $\mathcal{K}$ is a set of concepts that correspond to capacities and $\mathcal{T}_\mathcal{K}$ is a tree where nodes can be either a capability node (in $\mathcal{K}$) or a control node where ; denotes a sequence, + denotes a choice and // parallelism. Moreover, well-formedness of the tree imposes (i) leaves are capability nodes, (ii) capability nodes have at most one child, and (iii) control nodes have at least two children.

CSSs are inspired from GraphHTN trees yet, due to the targeted objective (service composition), we use different control nodes that correspond to the basic constructs of workflow languages. GraphHTN treatment of trees is simple as only concrete tasks (leaves) can be used in plans. Meanwhile, due to vertical adaptation, we do not know at which abstraction level the capacity of composed

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

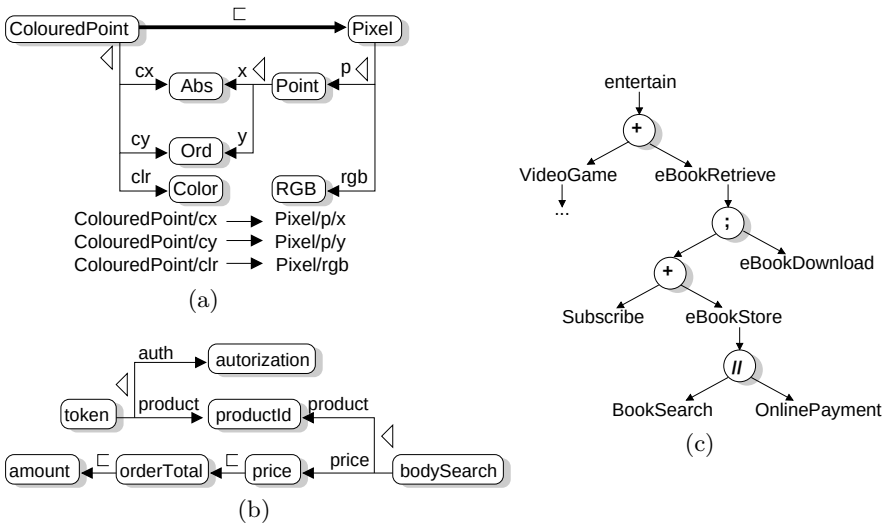

Fig. 7. CSS and DSS examples

services will be found, hence we have to consider all action nodes, not only leaves, *e.g.*, in Figure 7(c), if one looks for a eBookStore capability, we may either find directly a service with it, or we may compose (in parallel) services with capabilities BookSearch and OnlinePayment. Finally, concrete tasks in GraphHTN trees are associated with inputs and outputs (see Fig. 6). We remove this hypothesis as in service composition there may be several possible instances of an action node (several services with the corresponding capability), all with different sets of input and/or output data (*e.g.*, Paypal and MPS in Fig. 3).

A *service* is defined by its external view, *i.e.*, a conversation in YAWL. In order to be able to automatically compose services, we suppose each service has a (semantic) *capability*, and each operation defined in the service conversation has a set of (potentially empty) inputs and a set of (potentially empty) outputs. Finally, a *user task* corresponds to a service without conversation (the objective is to retrieve it). Hence, it is only given as a capability, a set of provided input data and a set of expected output data. In the sequel we suppose the user task is (eBookRetrieve, {title, customerId, creditCard}, {PRC}).

4.2 Adaptive graphplan algorithm

In this section we first introduce an adaptive planning technique which uses CSSs and extends GraphHTN to support vertical adaptation and multiple solution obtaining. In a second step, we extend this adaptive planning technique to support also horizontal adaptation thanks to DSSs. Finally, we present plan extraction and the encoding into YAWL.

Composition with vertical adaptation. There are important differences between GraphHTN trees and CSS, and in the way we use them for service composition in Task-Oriented Computing. While GraphHTN chains only concrete tasks (the tree leaves), we do not know at which abstraction level (in the CSS) services will be found, and several services may instantiate a capability. Moreover, we do not want to generate only the first possible plan but *all* possible service compositions. All these features yield algorithmic issues wrt. GraphHTN that will be explained in the sequel.

Service discovery and l-CSS computation. This step is used to transform the CSS structure to help reusing GraphHTN algorithms and to pre-select services that may potentially be used in the composition. For this, we compute a *labelled CSS (l-CSS)* as follows (see Fig. 8 for the application to Fig. 7(c)). First, we keep only the CSS subtree whose root is the capability corresponding to the user task which is supposed to be at least as abstract as the service capabilities. This is a sensible hypothesis in Task-Oriented Computing. Then, abstract capability nodes (*i.e.*, non-leaf ones) are replaced by a *method node* (M) which denotes a choice: it can be either instantiated directly by some service *or* its definition (*i.e.*, its sub-tree) can. In Figure 7(c), eBookStore may be either obtained by calling @eBook or by composing in parallel Amazon (capability BookSearch) and Paypal or MPS (capability OnlinePayment). Each capability node is replaced by the service that supports it (in case of several services as for OnlinePayment, the services are siblings under a M node). Finally, branches with no service instances are discarded and control nodes with only one child are reduced (*e.g.*, a branch $\mathcal{N} \rightarrow + \rightarrow \mathcal{T}$ is replaced by $\mathcal{N} \rightarrow \mathcal{T}$).

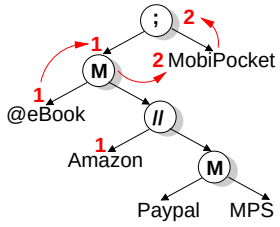

Fig. 8. l-CSS example (marking is used in graphplan building wrt. Fig. 9)

Graphplan building. The graphplan is built using and marking the l-CSS (see Figs. 8 and 9). Wrt. GraphHTN, ; and // nodes are treated as AND nodes, while M and + nodes are treated as OR nodes, and capability nodes as activity nodes. Algorithmic differences result from our multiple solution objective. First, there may be several services instantiating a capability (children of M nodes) that are possible at some step if all have their input data available. While including them all in the graphplan, we add exclusion constraints between them to ensure they

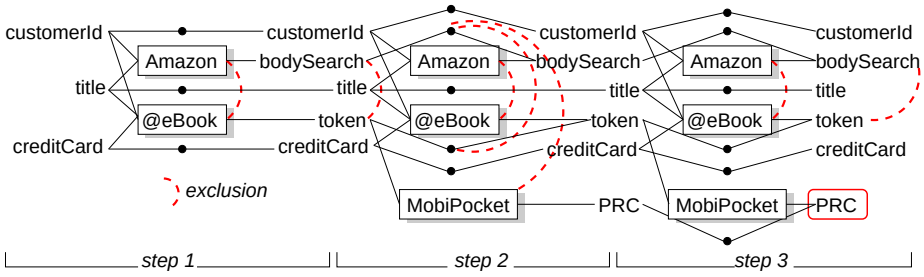

Fig. 9. Adaptive graphplan building (no data adaptation)

are exclusive in the solutions (*e.g.*, Amazon and @eBook). Moreover, we let a marked leaf be added again in the graphplan. This enables us to generate solutions where the corresponding service appears in different places. For example, this is mandatory to enable two children S_1 and S_2 of a // node to generate the $S_1;S_2$ and $S_2;S_1$ parts in two plans. Finally, with GraphHTN, the building algorithm stops once the root node of the search tree is marked, which generates only one plan. To be able to generate all possible plans we compute the length of the graphplan required to obtain them all using the following rules: (i) the length for a capability node is 1, (ii) the length for a ; or a // node is the sum of the lengths of its children nodes and (iii) the length for a M or a + node is the maximum of the lengths of its children nodes. We stop the building algorithm only when this length (here, 3) is reached.

Adding horizontal adaptation to the picture. Another important difference with GraphHTN is horizontal (data) adaptation. We present here how we extend the aforementioned technique to support this. Let us suppose we are after step 1 of Figure 9 and continue in Figure 10. Following the l-CSS (Fig. 8) where nodes @eBook and Amazon would be marked, MobiPocket (for eBookDownload), MPS and Paypal (for OnlinePayment) are theoretically applicable. Let us concentrate on the later one. Practically, Paypal is not applicable as its input data precondition on orderTotal is not satisfied. However, looking at the

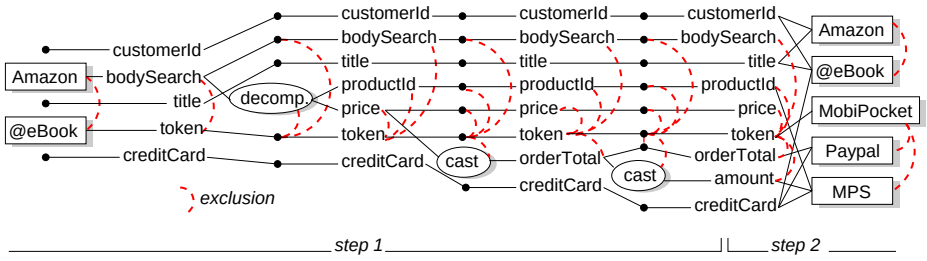

Fig. 10. Adaptive graphplan building (with data adaptation, principle)

DSS (Fig. 7(b)), we see that this can be obtained from price which in turn can be obtained using decomposition of bodySearch, which is available. The idea for horizontal adaptation is to add such data transformations as new actions layers in-between the ones that are computed with the previous algorithm. The possible transformations[4] are those presented with the CSS model in Section 4.1: decomposition, decomp(d,D) if $D = \{d_i \mid d \lhd d_i\}$), composition, comp(D,d) if $D = \{d_i \mid d \lhd d_i\}$), and casting, cast(d1,d2) if $d_1 \sqsubset d_2$. Interestingly one can have a task vision of these, *e.g.*, task cast above has precondition d_1 and postcondition d_2. This means that after each action layer of the previous algorithm, we may apply a forward chaining over such transformation tasks. The initial world is defined to be the last data layer (in Fig. 10, this is {customerId, bodySearch, title, token, creditCard}), while the data adaptation planning step is directed toward the set of data required for services that are theoretically applicable but are currently missing (in Fig. 10, this is {orderTotal} for Paypal and {amount, productId} for MPS). Note that since data available in the initial layer of this may be in mutual exclusion, rules for mutual exclusion apply also for horizontal adaptation. In our example, once orderTotal, amount and productId are obtained, the data adaptation step may end and we get back to service planning as presented before where Paypal and MPS are now applicable. The graphplan is further developed alternating horizontal and vertical techniques until requested data are available (here PRC) and the correct length (3 service layers) is reached. This graphplan is presented in Appendix.

Plan extraction and YAWL generation. As for techniques presented in Section 3, extraction is performed backtracking from the objective, here the user task expected data. Yet, contrary to GraphHTN, whose tree only supports graphplan building, we also use our l-CCS for on-the-fly filtering of plans while doing extraction. Filtering is required by the fact that, as presented above, to generate all compositions we have to be more permissive when constructing the graphplan.

Several plans are generated for our example:

- @eBook;MobiPocket,
- Amazon;decomp(bodySearch,{productId,price});cast(price,orderTotal);Paypal; comp({autorization,productId},token);MobiPocket, and
- Amazon;decomp(bodySearch,{productId,price});cast(price,orderTotal); cast(orderTotal,amount);MPS;comp({autorization,productId},token);MobiPocket.

However, to the contrary of task planning, the work is not done. We now have to transform plans into orchestrators written in an implementable model language, namely YAWL. This is demonstrated in Figure 11 for the second of the plans above.

The process we are defining has a single operation, whose name corresponds to the user task capability. The operation input and output data also correspond

[4] We omit composition attributes (x in $d_1 \lhd_x d_2$) here and in Fig. 10 graphplan for clarity reasons.

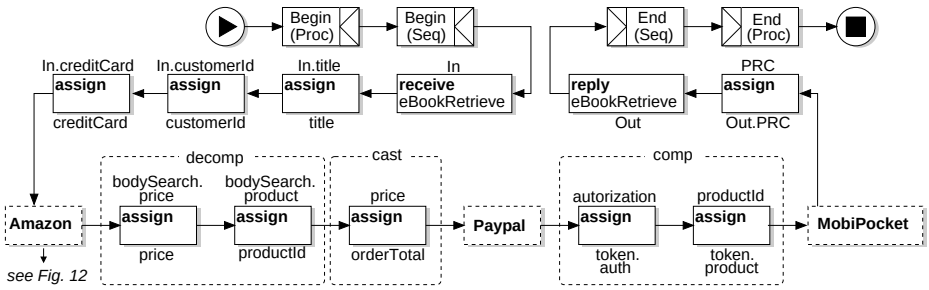

Fig. 11. A composition for user task (eBookRetrieve, {title, customerId, creditCard}, {PRC}) (YAWL)

to the user task ones. Variables are defined for all semantic data types (with the same name, *e.g.*, variable title of type title) and for messages (with a unique name built using the corresponding partner/service, operation name and message direction, *e.g.*, AmazonloginIn or AloginIn in Fig. 12 for place matters). Ordering and parallelism in the plans are respectively transformed into sequences and flows.

Actions in plans can be service capability use or horizontal adaptation features: cast, composition and decomposition. Casts are translated with assignments. To cast d_1 into d_2, we use as many assignments as there are $\rightsquigarrow$ rules for them, *e.g.*, 3 to cast a ColouredPoint into a Pixel in Figure 7(a). In case of simple casts between basic types, a single assignment is required. Composition of a set of d_i into some d is translated with one assignment for each d_i, *e.g.*, token.auth := autorization and token.product := productId. Decomposition is achieved in the reverse way.

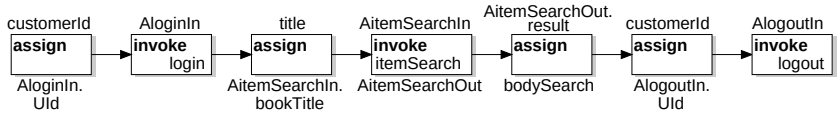

Fig. 12. Amazon conversation integration (YAWL)

Service capability use is achieved through conversation integration (see, *e.g.*, Fig. 12 for Amazon). To this purpose, we use the service conversation (*e.g.*, Fig. 2 for Amazon). We take the hypothesis that for two-way operations, the corresponding receive and reply activities are following in the conversation. Then, the conversation can be obtained using an invoke activity in the conversation integration for each receive activity in the service original conversation. The link between message parts of the invoke activities and orchestration variables is achieved using assignments.

Finally, we add a prologue and an epilogue to the YAWL workflow. The prologue corresponds to the beginning of the process and to the reception of a message for the user task operation. The epilogue corresponds to the reply relative to the aforementioned message reception and to the end of the process. The complete YAWL workflow is given in Appendix.

Tool. Our approach is fully automated thanks to GraphAdaptor, a tool written in Python. It takes as input a set of description files for user task, service, CSS and DSS. Additionally, for a service, the description file refers to a YAWL file. As a result, GraphAdaptor outputs a YAWL file for each possible service composition satisfying the user task.

5 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. Among these, planning-based approaches have particularly been studied due to their support for underspecified requirements [13–16]. As an alternative, automatic composition has also been achieved using matching and graph/automata-based algorithms [17–19] or logic reasoning [20]. Various criteria could be used to differentiate these approaches, yet, due to our Task-Oriented Computing motivation, we will focus on issues related to user task requirements, vertical, and horizontal adaptation.

While both data input/output and capability requirements should be supported to ensure composition is correct wrt. the user needs, only [15, 18] do, while [13, 14, 16, 17, 19] support data only and [20] supports capabilities only. As far as adaptation is concerned, [14, 16, 18, 19] support a form of horizontal (data) adaptation, using semantics associated to data; and [13] a form of vertical (capability abstraction) adaptation, due to its hierarchical planning inheritance. In our proposal, we combined both techniques to achieve both adaptation kinds.

Some approaches support more expressive models in which protocols can be described over capabilities – either for the composition requirement [20] or for both composition and services [15, 18]. Up to now, like [13, 14, 16, 17, 19], we only support conversations over operations (for a given capability). Yet, we advocate that in most cases this is not a limitation for the user task, that can be described with a single abstract capability meanwhile our hierarchical planning tree (CSS), with its workflow-inspired operators (nodes), can be used to specify more concrete capability protocols.

As opposed to the aforementioned works dealing with orchestration, in [21], 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.

Few works explicitly add adaptation features to Service-Oriented Computing [4, 5]. They adopt a different and complementary view wrt. ours since their objective is not to integrate adaptation within service composition in order to increase the orchestration possibilities, but rather to tackle protocol adaptation between clients and services, *e.g.*, to react to service replacement. Indeed, the more advanced software adaptation works [22–24] are model-based approaches whose objective is to solve protocol mismatch between a fixed set of components, and do not tackle the discovery of the required components nor the composition towards user needs. Rather, they support a form of composition verification, *e.g.*, through temporal logic requirements (defined at the same abstraction level than the composed entities) for the composition. Moreover, but for [22], where adaptation is applied to COM/DCOM components, they do not address adaptor implementation. While these works rely on simple labelled transition systems models, we rely on YAWL which is closer to the services models, hence made it possible to derive WS-BPEL code in a simpler way, following [4].

6 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 hierarchical planning. We are also able to generate different composition solutions to the user task requirement, while ensuring they are correct from both data and semantics points of view.

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. A first perspective concerns this issue. Upon disappearance of a service to be used in the future (wrt. the composition step currently executed), the graphplan – that contains more than one composition solution – could be used to make service replacement easier. A second perspective concerns service model expressiveness. Protocols could be defined over capabilities, as in [15, 18], to enable the composition of more complex services. Finally, a longer term perspective concerns another extension of our service model, using security features in order to support the automatic composition, correct by construction, of complex user tasks.

References

1. Papazoglou, M.P., Georgakopoulos, D.: Special Issue on Service-Oriented Computing. Communications of the ACM **46**(10) (2003)
2. Canal, C., Murillo, J.M., Poizat, P.: Software Adaptation. L'Objet **12** (2006) 9–31

3. 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. Volume 3938 of LNCS. (2006)
4. Brogi, A., Popescu, R.: Automated Generation of BPEL Adapters. In: Proc. of ICSOC'06. (2006)
5. Motahari-Nezhad, H.R., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-Automated Adaptation of Service Interactions. In: Proc. of WWW'07. (2007)
6. Peer, J.: Web Service Composition as AI Planning – a Survey. Technical report, University of St.Gallen (March 2005)
7. Marconi, A., Pistore, M., Poccianti, P., Traverso, P.: Automated Web Service Composition at Work: the Amazon/MPS Case Study. In: Proc. of ICWS'07. (2007)
8. van der Aalst, W., ter Hofstede., A.: YAWL: Yet Another Workflow Language. Information Systems **30**(4) (2005) 245–275
9. Blum, A.L., Furst, M.L.: Fast Planning Through Planning Graph Analysis. Artificial Intelligence **90**(1-2) (1997) 281–300
10. Erol, K., Hendler, J., Nau, D.: Semantics for Hierarchical Task Network Planning. Technical Report UMIACS-TR-94-31, University of Maryland at College Park (1994)
11. Lotem, A., Nau, D.S., Hendler, J.A.: Using Planning Graphs for Solving HTN Planning Problems. In: Proc. of AAAI/IAAI'99. (1999)
12. Euzenat, J., Shvaiko, P.: Ontology Matching. Springer (2007)
13. 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)
14. Constantinescu, I., Binder, W., Faltings, B.: Service Composition with Directories. In: Proc. of SC'06. (2006)
15. Pistore, M., Traverso, P., Bertoli, P., Marconi, A.: Automated Synthesis of Composite BPEL4WS Web Services. In: Proc. of ICWS'06. (2006)
16. Liu, Z., Ranganathan, A., Riabov, A.: Modeling Web Services using Semantic Graph Transformation to Aid Automatic Composition. In: Proc. of ICWS'07. (2007)
17. Brogi, A., Popescu, R.: Towards Semi-automated Workflow-based Aggregation of Web Services. In: Proc. of ICSOC'05. (2005)
18. 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) (2007) 1941–1955
19. Benigni, F., Brogi, A., Corfini, S.: Discovering Service Compositions that Feature a Desired Behaviour. In: Proc. of ICSOC'07. (2007)
20. 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'04. (2004)
21. Melliti, T., Poizat, P., Ben Mokhtar, S.: Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services. In: Proc. of FASE'08. (2008)
22. Inverardi, P., Tivoli, M.: Deadlock Free Software Architectures for COM/DCOM Applications. Journal of Systems and Software **65**(3) (2003) 173–183
23. Bracciali, A., Brogi, A., Canal, C.: A Formal Approach to Component Adaptation. Journal of Systems and Software **74**(1) (2005) 45–54
24. Canal, C., Poizat, P., Salaün, G.: Model-based Adaptation of Behavioural Mismatching Components. IEEE Transactions on Software Engineering **34**(4) (2008) 546–563

A Additional Figures

see next pages

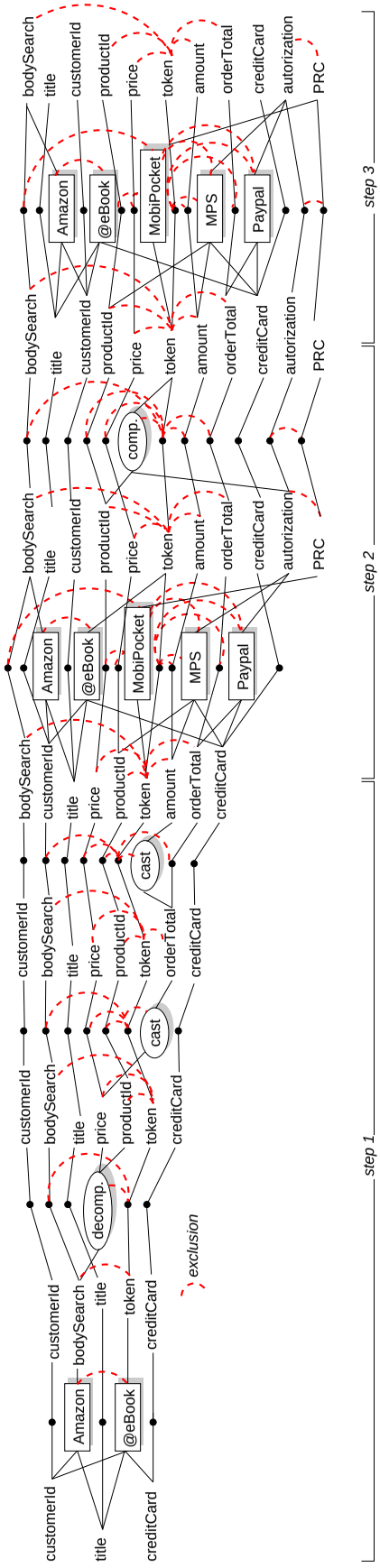

Fig. 13. Adaptive graphplan building

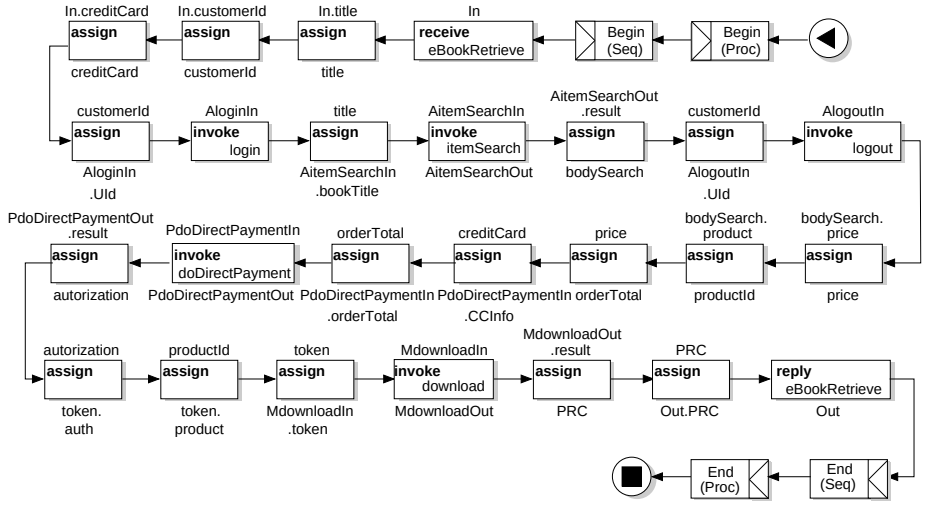

Fig. 14. A composition for user task (eBookRetrieve, {title, customerId, creditCard}, {PRC}) (YAWL)