SEMANTIC ORCHESTRATION MERGING
Towards Composition of Overlapping Orchestrations

Clementine Nemo, Mireille Blay-Fornarino, Michel Riveill
I3S Laboratory, Rainbow team
CNRS - University of Nice-Sophia-Antipolis
930, route des colles, 06903 Sophia-Antipolis

Günter Kniesel
Computer Science Department III, ROOTS group
University of Bonn
Römerster. 164, D-53117 Bonn, Germany

Keywords: Service Oriented Architecture (SOA), composition, merge, transformation.

Abstract: Service oriented architectures foster evolution of enterprise information systems by supporting loose coupling and easy composition of services. Unfortunately, current approaches to service composition are inapplicable to services that share subservices or data. In this paper, we define *overlapping orchestrations*, analyse the problems that they pose to existing composition approaches and propose *orchestration merging*, a novel, interactive approach to composition of overlapping orchestrations based on their semantic.

1 INTRODUCTION

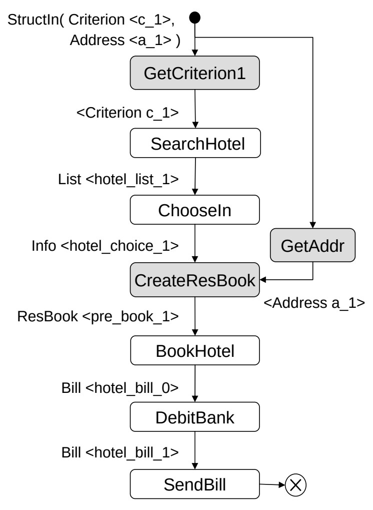
The need to adapt enterprise information systems (EIS) to ever changing requirements calls for software architectures that allow to master complexity without preventing evolution. Service-oriented architectures are particularly well-suited for modern EIS. They foster evolution by supporting loose coupling among services and allowing easy composition of new services from existing ones. Services that do not call themselves other services are called *basic services* (Bartoli et al., 2005). A *service orchestration* defines a *composite service* from several other services.

An orchestration encapsulates knowledge about how to handle a task, the adaptations required to integrate services with mismatching interfaces and the protocols of the integrated services. The ability to expose an orchestration as a service enables recursive composition, which in turn enables inter-workflow interaction, higher levels of reuse and additional scalability (Khalaf et al., 2003).

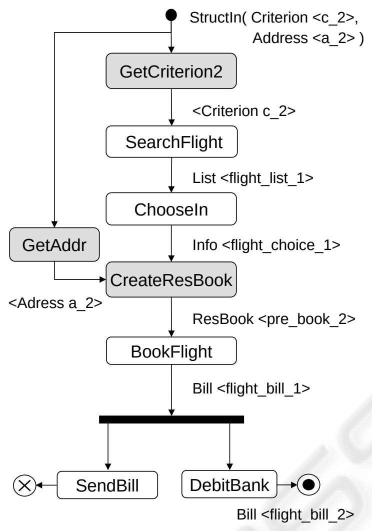
Elaborating a service orchestration is very demanding in terms of effort and domain knowledge. Therefore, different systems have been proposed to ease the programmer's tasks For instance, Oracle BPEL Designer (Chandran and Poduval, 2005) enables completely visual specification of orchestrations for basic services. The ADAPT framework (Bartoli et al., 2005) additionally supports automated matching of basic service parameters to composite service parameters with the same name. For other parameters, dataflow relations are visually specified by the programmer. (Kazhamiakin et al., 2006). For a given orchestration, Oracle BPEL Designer, Adapt and the approach of Kazhamian et al. (Kazhamiakin et al., 2006) support the verification of different safety and liveness properties. Whereas the above approaches are confined to orchestration of basic services, Khalaf et al (Khalaf et al., 2003) support orchestration of composite services.

Unfortunately, all known approaches to orchestration composition are inapplicable to *overlapping orchestrations*, that is, to orchestrations that use common services or share data. In this context, our paper provides the following contributions :

- definition of overlapping orchestrations (Sec. 2),
- illustration of the need to compose overlapping orchestrations,
- explanation of the problems of existing approaches in the presence of overlapping orchestrations,
- introduction of *orchestration merging*, an alternative to traditional composition approaches that is applicable to overlapping orchestrations (Sec. 3).
- introduction of model transformation rules that guide the merging process (Sec. 4).



a) HBO: Hotel Booking Orchestration.



b) FBO: Flight Booking Orchestration.

Figure 1: Initial hotel booking orchestration and flight booking orchestration.

2 PROBLEM STATEMENT

In this section we introduce our representation of orchestrations, define overlapping orchestrations, demonstrate the need for composing them by introducing a running example and discuss the problems that they pose to known composition approaches.

2.1 BPEL Orchestrations

In this paper, we focus on orchestrations as defined by the OASIS Consortium (MacKenzie et al., 2006). The Business Process Execution Language for Web Services (BPEL4WS) is the principal standard defining orchestrations. It supports *primitive activities* to receive, return, read, assign, and modify data, invoke a service, terminate the orchestration, wait or throw an exception. In order to combine primitive activities BPEL supports *structured activities* allowing sequential, concurrent and conditional execution of services. Orchestrations built from BPEL activities are described in the following by UML activity diagrams. For simplicity, we regard an orchestration as a function with a single parameter. This is no restriction since a set of parameters can always be replaced by a single parameter whose value is the aggregation of individual parameter values. The aggregation of values and the selection of subcomponents from an aggregate are represented as explicit adaptation activities, marked by gray boxes in the diagrams. For instance, in Fig.1a) *GetAddr* selects the address part of the orchestration's input and *CreateResBook* generates an aggregate from the address and the hotel choice information.

2.2 Running Example

We use the classic example of a travel agency as our reference use case (Kazhamiakin et al., 2006). A travel agency offers booking of flights or hotels. The booking system consists of two orchestrations. The first describes the hotel booking process (Fig. 1a) and the flight booking process (Fig. Figure 1b).

We explain in detail the first example:

- The Hotel Booking Orchestration (HBO) takes an input that consists of two data elements with different types. The *Criterion* variable encapsulates information used to select hotels. The *Address* variable represents the address of the customer.
- The HBO is started by extracting the *Criterion* value from the input and passing it as a parameter to an invocation of the *SearchHotel* service.
- With the list of bookable hotels returned by *SearchHotel* the *ChooseIn* service is invoked to let the user select a hotel.
- The selected hotel and the customer address are aggregated and passed to the *BookHotel* service.
- Finally, the process invoices the payment to the customer's bank and sends him the bill. This is done by invoking first the *DebitBank* service and passing the returned *Bill* to the *SendBill* service (which has no return value).

The flight booking orchestration (Figure 1b) only differs in the objects on which it works (flights instead of hotels) and in that *SendBill* and *DebitBank* are executed concurrently. The orchestration returns the *Bill* produced by *DebitBank*.

2.3 Overlapping Orchestrations

We say that two *orchestrations overlap* if they share services or would share input or output data after composition. In the travel agency example, the two presented orchestrations overlap because they share the services *ChooseIn*, *DebitBank* and *SendBill* and because in a composition they would share the customer address.

Now assume that the travel agency wants to jointly offer flights *and* hotels to its customers. For this, it needs to have an integrated process for booking flights and hotels. Implementation of such a process either requires writing a completely new service (which is certainly undesirable) or the ability to compose the existing services.

Unfortunately, known approaches to service composition are inapplicable in our example. They would treat the existing orchestrations as black boxes that are executed either sequentially or in parallel. In both cases, black-box reuse of the existing orchestrations would lead to highly undesirable effects:

- The payment service (DebitBank) would be called twice for the same journey, resulting in double as high network traffic and twice as long time for payment authorization.
- Separate processing of hotel and flight payment might result in authorization of the first and rejection of the second payment if the customer's account balance is insufficient for paying both. Such cases need further invocations of the payment service for revoking the already performed first payment. In addition to the added costs and network traffic, their treatment complicates the task of the programmer who specifies the orchestration, requiring complex additional code for transaction management.
- In the case of direct debit payments, which are preferred by many companies because they only involve a relatively small, fixed fee per transaction the double service invocation will result in double online payment costs.
- Two separate bills would be sent to the customer, increasing the postage costs of the company and confusing the customer. If the bills do not arrive simultaneously, customers might believe that the booking not shown on the received bill had failed.

The problem illustrated by this small example is that black-box reuse prevents identification and proper treatment of shared data and shared services. For instance, in our example one would need to send just one *joint* bill to the customer and ask the online payment service just once for the *total* sum of the journey.

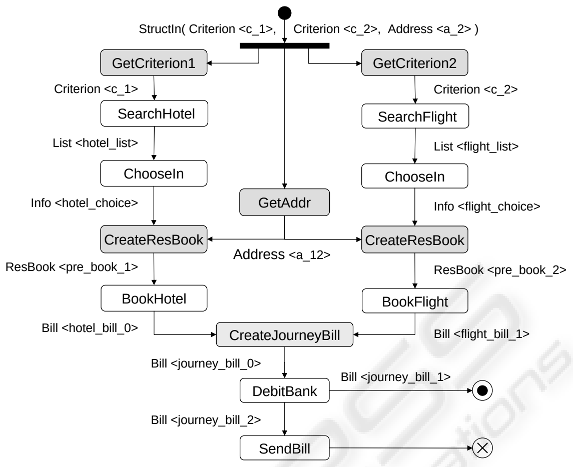


Figure 2: Merged Journey Booking Orchestration (JBO).

3 ORCHESTRATION MERGING

When confronted with overlapping orchestrations without proper tool support, developers typically create a new orchestration in which they integrate the initial orchestrations via a series of routine transformations. The application of the following standard actions on our example yields the merged orchestration illustrated in Figure 2:

(i) Unifying input data: Input parameters that represent the same data are unified. In our example, the programmer determines that this is the case for the *Address a_1* and *Address a_2* components of both inputs (Figure 1). Therefore they are unified in Figure 2. The search *Criterion c_1* and *Criterion c_2* components for hotels and flights are different, so both are kept. In order to have only one input parameter, a structure is created to encapsulate *a_1, c_1* and *c_2*.

(ii) Detecting multiple calls to the same service: If the same service is invoked on the same data and the result is assigned to the same (or a unified) variable, then the invocations can be unified. If the input or output variables are different, the programmer can still decide that they should be unified. For instance, in our example, the programmer decides to send just one *joint* bill to the customer and ask the online payment service just once for the *total* sum of the journey. Therefore she merges the flight and hotel bills using the *CreateJourneyBill* adapter. However, the two calls to the *ChooseIn* service are kept since their input data and output data is different and must not be unified.

(iii) Preserving the partial order of instructions: The order of instructions in the resulting orchestration has to respect the orders of the input orchestrations. The hotel booking orchestration (Fig. 1a) imposes an order between *DebitBank* and *SendBill*. The flight booking orchestration (Fig. 1b) in the resulting orchestration, these two invocations are executed in the order specified for hotel booking.

(iv) Unifying output data: In order to be correctly defined, the resulting orchestration must return a single output data element. Therefore, the developer has to unify the output parameters or aggregate them into a joint structure. This is similar to point (i). This case does not occur in our example since one of the input orchestrations returns no result.

3.1 Interactive Orchestration Merging

With the current state of the art, developers perform all of the above steps manually when confronted with the need to compose overlapping orchestrations. They inspect the BPEL specification of the input orchestration, identify shared data and shared service calls and manually create the integrated orchestration. Some environments ease understanding BPEL code by providing graphical notations (Bartoli et al., 2005), (Ben Mokhtar et al., 2006). However, this is insufficient since the identification of overlaps and the detection of unification candidates, the decision about unification and the merge process itself is still left to the programmers. They have to repeat these steps after every change of the input orchestration, even if none of the sharing relations has been modified. This is not only a waste of human resources but also a source of errors that can be injected in the course of the manual process.

One could consider automated orchestration merging as an alternative. However, full automation is infeasible unless semantic specifications of each service and data item are available. Without them we cannot deduce, for instance, how to compose the two calls to the *ChooseIn* service. Therefore, we propose an *interactive* orchestration merging process that indicates potential merging points to developers, remembers their decisions for later reuse, automates the unification of invocations or data items decided by the programmers in order to eliminate potential sources of errors and automatically unifies input and output data whose conceptual identity can be derived from previous unification steps.

3.2 Merge Process

The interactive merge process illustrated in Figure 3 comprises three phases: the creation of an orchestration model, the transformations of the orchestration model and the reverse engineering of BPEL from the model.

From BPEL to an Orchestration Model (OM): In the first step, the BPEL specification of each input orchestration is transformed to an internal representation. This phase corresponds to the transformation T_{in} in Figure 3. It verifies and normalizes the input. Variables are renamed to avoid accidental name clashes, as shown in Figure 1a) and 1b). Multiple basic invocations of the same service are grouped in a structure named *complex invocation*. We verify that each branch of an alternative has at most one reply. These preparations simplify the later steps.

Guiding the developer throughout the merging of OMs: The merging tool explores all the orchestration models given as inputs and detects required decision points. Decision points can be potential merge points or problem cases. For instance, concurrent reading and writing of the same variable is a problem case that requires the programmer to decide about an order of these operations. Alternatively, two variables with the same type in invocations of the same service could be candidates for a unification. Developers choose to unify the merge candidates or mark them as semantically different. The transformation process automatically computes the resulting orchestration and updates the remaining merging points. After every transformation, the process verifies that the partial order of instructions from the input orchestrations is still preserved. For example, if in some block of an input orchestration the service A is invoked before the service B, the corresponding block of the resulting orchestration must not invoke the service B before A. In addition, the precedence relationship must not contain cycles introduced by unification of two variables. If a cycle is detected the last merge is undone and the programmer is asked to revise his related decision.

Back to BPEL: The merge is finished when the resulting orchestration is a function with one input and one output, and all shared calls are unified or marked as different. When this final state is reached the result model is transformed back to BPEL with the transformation T_{out} shown in Figure 3.

Figure 3 shows the global process in which transformation rules implement the steps explained above and guide the actions of programmers. The transformation rules are detailed in Section 4.

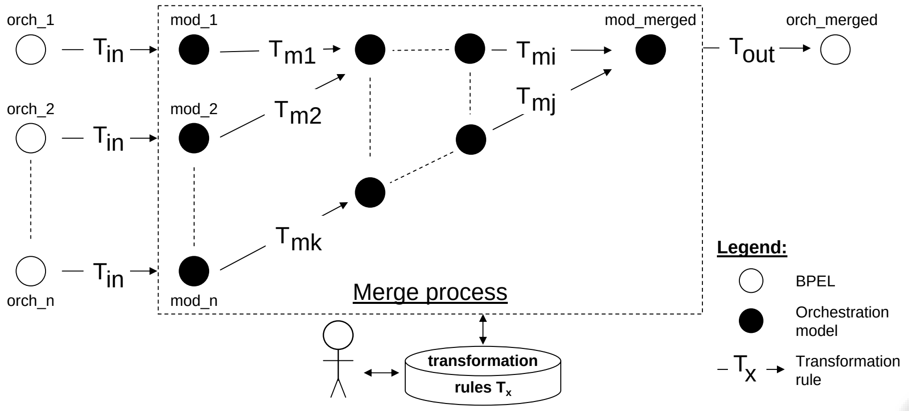


Figure 3: Interactive merge process guided by transformation rule execution.

4 TRANSFORMATION RULES FOR ORCHESTRATION MERGING

We define an *orchestration* as a tuple *o = (V, I, prec, cond)* where *V* is a set of variables described by a name and a type, *I* is a finite set of instructions, *prec* is a function that returns the list of instructions preceding a given instruction and *cond* is a function that returns a list of conditions that must be true before executing a given instruction. A 'switch' statement is expressed by a set of mutually exclusive conditions.

An *instruction* is characterized by an identifier, an ordered list of input variables, an ordered list of output variables and a primitive activity. A primitive activity is a basic invocation, a return or an adapter. A complex invocation groups several basic invocations to the same service in order to normalize orchestrations. It is characterized by the name of the invoked service, the set of basic invocations to this service and a composition function that defines how the results of basic invocations are related (Montagnat et al., 2006).

In order to define and analyze[1] transformations of orchestrations, we use an orchestration model and define transformation rules on this representation. We adopt the logic-based model representation and the conditional transformation formalism developed by Kniesel (Kniesel and Koch, 2004; Kniesel, 2006). For lack of space, we do not describe the formal details of the representation but confine ourselves to an informal description of the transformation rules that guide the developer in creating new orchestrations:

- *At most one input variable*: This rule detects a potential merge point when an input variable is free, i.e. unconnected to the output of any instruction. The programmer is asked to formulate relationships between free variables. Depending on the variable types, different relationships can be expressed. Variables with identical types can be unified (e.g. type Address in our example). If the value of a variable is contained in another one, the variables are in an inclusion relationship. For instance, the bill identifier is contained in the variable denoting the bill. In this case, an adapter is added with the Bill variable as input and the Identifier variable as output. If none of these relationships hold between the input variables *(v1...vn)*, we generate a new variable *(v)* whose type aggregates the input variables types. This new variable is the input of the merged orchestration. We generate adapters that yield the values of each former input variable *(v1...vn)* by accessing the corresponding component of *(v)*.
- *A single return in each branch* : When an output is expected, there must not exist more than a single returned element for each branch. As in the previous rule, if we detect that there exist several return instructions, the developer must specify relationships between these instructions, e.g. mathematical operations that combine their results. Variables are then composed or some return instructions are removed.
- *Unifying basic invocations* : This rule detects a potential merge point when there are several basic invocations to the same service and when these invocations *(i1...in)* are not part of a complex invocation. The user can either unify them, or create a complex invocation. In the first case, an adapter can be used to unify input variables. Output variables are unified automatically. A new basic invo-

[1]Describing the use of Condor (Kniesel and Bardey, 2006) for analysing dependencies between transformation rules is outside the scope of this paper.

cation *i* is created that is subject to all the guarding conditions of the unified instructions *(i1...in)*. In the second case, invocations *(i1..in)* have to be separately executed : a complex invocation *(i)* is created by the user who specifies the composition function. Each basic invocation *(i1..in)* references the complex invocation *(i)*. The functions *cond* and *prec* concerning the orchestration remain the same.

- *Unifying complex invocations* : This rules considers the need for a merge when a complex invocation to a service *A*, and another invocation (basic or complex) to the same service *A* are detected. The developer must specify the result of the merge as a new complex invocation that is referenced by all the merged basic invocations.
- *Unifying adapters*: When several instructions use the same adapter with the same input variable, this rule proposes to the user to unify these instructions and subsequently also the output variables.

5 CONCLUSION AND PERSPECTIVES

In this paper we addressed the need to broaden the applicability of orchestration composition to overlapping orchestrations, which share services or parameters. We demonstrated that traditional black-box composition of orchestrations is inappropriate in the case of overlapping orchestration because it fails to avoid redundant or erroneous multiple invocations of the same service. Therefore, programmers facing the task to combine overlapping orchestrations must currently perform a tedious and error-prone manual process.

As a remedy, we presented orchestration merging, an interactive, computer supported process that guides programmers step-by-step through the integration task, automating many subtasks. Orchestration merging identifies potential merge points and gives programmers a chance to to unify them or mark them as distinct. If programmers decide to merge, the system automates the merging steps. After each transformation it checks the consistency of the orchestration model and computes remaining merging points. When errors are detected, developers can undo preceding merging steps. We believe that our approach is a contribution to more reliable and easy to evolve service-oriented systems.

Besides the ongoing implementation of our approach there are different interesting conceptual extensions. Continuous evolution of orchestrations by substitution, addition and deletion of services suggests the need to integrate verification of service compatibility (Martens, 2005), substitutability of services (Camara et al., 2005) and equivalence of services (Ben Mokhtar et al., 2006).

REFERENCES

Bartoli, A., Jiminez-Peris, R., Kemme, B., Pautasso, C., Patarin, S., Wheater, S., and Woodman, S. (2005). The ADAPT framework for adaptable and composable web services. *IEEE Distributed Systems Online*, 6(9).

Ben Mokhtar, S., Geogantas, N., and Issarny, V. (2006). COCOA : Conversation-Based Service Composition for Pervasive Computing Environments. In *IEEE International Conference on Pervasive Services (ICPS)*, Lyon (France).

Camara, J., Canal, C., Cubo, J., and Vallecillo, A. (2005). Formalizing WSBPEL Business Processes Using Process Algebra. In *Foundations of Coordination Languages and Software Architectures (FOCLASA)*, San Francisco (CA). Springer.

Chandran, P. and Poduval, A. (2005). Adding BPEL to the Enterprise Integration Mix. Technical report, ORACLE.

Kazhamiakin, R., Pistore, M., and Santuari, L. (2006). Analysis of communication models in web service compositions. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 267–276, New York, NY, USA. ACM Press.

Khalaf, R., Mukhi, N., and Weerawarana, S. (2003). Service-Oriented Composition in BPEL4WS. In *International World Wide Web Conference (WWW)*, Budapest (Hungary). W3C.

Kniesel, G. (2006). A Logic Foundation for Conditional Program Transformations. Technical report IAI-TR-2006-01, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany.

Kniesel, G. and Bardey, U. (2006). An analysis of the correctness and completeness of aspect weaving. In *Proceedings of Working Conference on Reverse Engineering 2006 (WCRE 2006*, pages 324–333. IEEE.

Kniesel, G. and Koch, H. (2004). Static composition of refactorings. *Science of Computer Programming (Special issue on Program Transformation)*, 52(1-3):9–51. http://dx.doi.org/10.1016/j.scico.2004.03.002.

MacKenzie, M., Laskey, K., McCabe, F., Brown, P., and Metz, R. (2006). Reference Model for Service Oriented Architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS.

Martens, A. (2005). Simulation and equivalence between bpel process models. In *Design, Analysis, and Simulation of Distributed Systems Symposium*, San Diego (California).

Montagnat, J., Glatard, T., and Lingrand, D. (2006). Data composition patterns in service-based workflows. In *Workshop on Workflows in Support of Large-Scale Science (WORKS'06)*, Paris, France.