Web Services Orchestration Evolution : A Merge Process For Behavioral Evolution

Sébastien Mosser, Mireille Blay-Fornarino, Michel Riveill

Web services orchestrations evolution:
A merge process for behavioral evolution

Sébastien Mosser, Mireille Blay–Fornarino, and Michel Riveill

University of Nice Sophia – Antipolis
Cnrs, I3s Laboratory, Rainbow team
Sophia Antipolis, France
{mosser,blay,riveill}@polytech.unice.fr

Abstract. Services Oriented Architectures preach loosely-coupled services and high–level composition mechanisms, using for example Web Services to define services and Orchestrations to compose them. But orchestration evolutions imply modification at source code level. This article shows how the orchestration paradigm itself can be used to support evolution of Web Services Orchestrations through a behavioral merge process. Using the same model to express orchestrations and evolutions, we expose formally and illustrate in this contribution a merging process helping Wsoa administrators to deal with behavioral evolutions.

1 Introduction

Services Oriented Architectures (Soa) [1] use the concept of *service* as an elementary brick to assemble complex systems. Services are loosely–coupled by definition, and complex services are build upon basics ones using compositions mechanisms. The loose coupling methodology enables the separation of concerns and helps systems evolution.

Using Web Services as elementary services, and Orchestrations [2] as composition mechanism, Web Service Oriented Architectures (Wsoa) provides a way to implement these loosely–coupled architectures. W3c define orchestrations as "*the pattern of interactions that a Web Service agent must follow in order to achieve its goal*" [3]. Specialized (*i.e. elementary*) code is written inside Web Services, and business processes are described as an orchestration of those Web Services.

Code manipulations, like the refactoring operation, help software evolution support. In [4], authors identify some challenges for future research on software evolution and focus on the abstraction need. Lehman identifies as his first *"Law of Software Evolution"* [5] that *"A program that is used must be continually adapted else it becomes progressively less satisfactory"*. As Wsoa focus on business reactivity and eternal adaptation to fit with market and anticipate trends, this well–known law makes sense twenty four hours a day.

This contribution deals with Wsoa orchestrations evolutions, focusing on behavioral evolutions. Our originality is to use the same model to represent the

behavior of orchestrations and evolutions. We propose a merging algorithm build upon this formal model helping integration of evolutions into orchestrations.

We identify in Sect. 2 the need of evolution capabilities inside orchestrations. Sect. 3 proposes a high level model for orchestrations supporting evolution reasoning and Sect. 4 shows how this model works on an example. Sect. 5 exposes validation of this work, Sect. 6 discusses related work about orchestrations evolution mechanisms. Finally, Sect. 7 concludes this paper and shows perspectives of our contribution.

2 Orchestration behavioral evolutions

Following the W3C definition, orchestrations can be represented as a *"white–box service"*: *"service"* because it basically defines a public interface (including data types) and *"white–box"* means that we can understand the behavior of such services, defined as the exchange of messages between other services.

Obviously, orchestrations can evolve in three ways: *(i)* interface, *(ii)* data type and *(iii)* behavior. This study focuses on static behavioral changes, *i.e.* the evolution process (presented here in Sect 3) handles orchestrations and evolution in a non–production state. Interface and data types evolutions are out of the scope of this paper, and refer more to refactoring [6] and model checking concepts.

In [7], authors sketched a taxonomy dealing with software changes and evolution. Using this taxonomy, we express in Tab. 1 the kind of evolutions this contribution deals with: our goal is to propose a partially–automated evolution process using a high–level reasoning abstraction.

Temporal Properties (*When*)			
Time of change:	*static*	**Change history:**	*parallel*
Change frequency:	*periodically*	**Anticipation:**	*unanticipated*
Object of Change (*Where*)			
Artifact:	*orchestration*	**Granularity:**	*coarse–grained*
Change propagation:	*traceable*	**Impact:**	*local*
Change Support (*How*)			
Degree of Automation:	*partially–automated*	**Change Type:**	*semantic*
Degree of Formality:	*partial*		–

Table 1. WSOA evolution using [Buckley *et al*, 2005] taxonomy

Example: We consider here an application called SEDUITE, based on a WSOA. This software uses different atomic services as *information sources* and users access to information using an orchestration called `InfoProvider` (Fig. 1). It describes a basic business process: using an authorization `ticket` and a user

`profile` as input data, it will return informations as `result` from a *source* called `News` in accordance with `profile`, if the given `ticket` is a valid ticket. Two kinds of evolutions commonly encountered[1] can be illustrated through the SEDUITE example:

- How to add a new source of information service into `InfoProvider` (*e.g.* a weather forecasting service, a calendar service, events notification, restaurant menu, TV shows, ...).
- How to ensure that a given input profile is correct (not empty, conform with current usage of the application, ...) before invoking sources ?

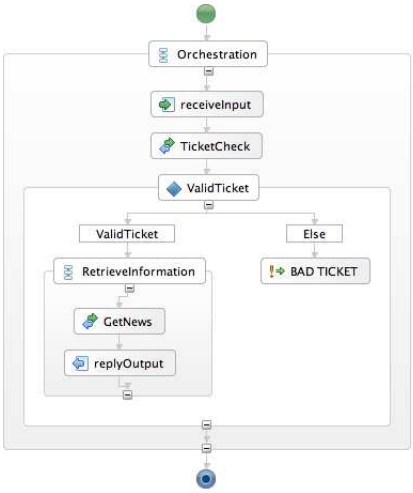


Fig. 1. `InfoProvider` orchestration using ECLIPSE BPEL Designer

Obviously, all these kinds of behavioral evolutions can be done at the orchestration language level by editing the source code (*e.g.* adding some activities, conditional statements, exceptions, ...) but this process is error-prone and off–putting.

Moreover, when the user wants to apply n different evolutions into the same orchestration (*e.g.* adding two new sources of informations), she can expect from the system some support mechanisms to ease the task. Our goal is to automate this process, providing an evolutions merging algorithm. This algorithm lets the user focus on evolution interactions and semantic. Next sections focus on these points, proposing an orchestration model able to merge evolutions.

[1] More informations and use cases at `http://anubis.polytech.unice.fr/jSeduite`

3 A high level reasoning model: "ADORE"

To perform high level composition and to allow reasoning on orchestrations and evolutions, we define a model called ADORE : *"Activity moDel suppOrting oRchestration Evolution"*. This section describes formally this model, and shows on the SEDUITE example how we can express an existing orchestration using it. It also describes the meaning of the $Merge$ operation enabling our process.

3.1 ADORE formalism: Orchestrations & Evolutions

Orchestration: An orchestration is a tuple $(A^\star, \prec^\star)$ representing a behavior. $A^\star$ is a set of *Activity* $\{a_1, \ldots, a_n\}$ and $\prec^\star$ a partial ordering between these activities.

Activity: An activity is a tuple $(uid, K, V^\star_{in}, V_{out}, G^\star)$. Each activity is unique inside an *Orchestration* and identified by *uid*. K refers to the *Kind* of this activity, $V^\star_{in}$ (resp. V_{out}) represent inputs (resp. output) *Variables* (identified by name). Constants are represented as variables with immutable content. An activity can take multiple input variables $\{in_1 \ldots in_n\}$ but returns exactly one result V_{out} (possibly $\emptyset$). $G^\star$ represents conditional guards and allows conditional expressions (`if/then/else`).

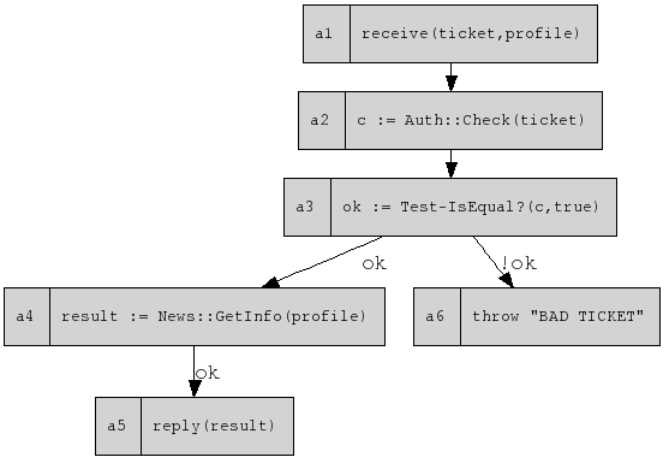
Partial ordering ($\prec$, precedence rules): Activities are ordered using an operator $\prec$. The expression $a_1 \prec a_2$ is called a precedence rule and means that a_2 must wait the end of a_1 to start its own execution. As our algorithm is based on acyclic behavior, we do not allow loop expressiveness for now.

Kind: We use in that model a subset of BPEL specifications [8]. We consider the following kind of allowed activity: *(i)* variable assignment ($assign(function)$), *(ii)* service invocation ($invoke(Service, Operation)$), *(iii)* message reception ($receive$), *(iv)* response sending ($reply$) and *(v)* fault report ($throw$). To deal with conditional statement, we add a *test* activity (a *test* activity evaluate a boolean predicate. It has exactly one output variable).

Guards: Guards refers to the expected *test* activity, and add a "true or false" semantic into our model. Adding $guard(a_t, true)$ as a guard on an activity a means that a will start only if a_t (which is a *test* activity) output is evaluated to true. Implicitly, it exists a precedence rule $a_t \prec a$.

Fig. 2 represents the `InfoProvider` orchestration using both textual and graphical formalisms (inspired by UML activity diagrams).

Evolution: An *Evolution* can be considered as a piece of orchestration which can be plugged into existing orchestrations. Evolution is therefore as a superset of orchestrations. $A^\star$ contains exactly one *hook* special activity which represent where the evolution will be connected into an orchestration.



$$O = (\{a_1, \ldots, a_6\}, \{a_1 \prec a_2, a_2 \prec a_3, a_3 \prec a_4, a_4 \prec a_5, a_3 \prec a_6\})$$
$$a_1 \equiv (a_1, receive, \{ticket, profile\}, \emptyset, \emptyset)$$
$$a_2 \equiv (a_2, invoke(Auth, Check), \{ticket\}, c, \emptyset)$$
$$a_3 \equiv (a_3, test(isEqual), \{c, true\}, ok, \emptyset)$$
$$a_4 \equiv (a_4, invoke(News, GetInfo), \{profile\}, result, \{guard(a_3, true)\})$$
$$a_5 \equiv (a_5, reply, \{result\}, \emptyset, \{guard(a_3, true)\})$$
$$a_6 \equiv (a_6, throw, \{\text{“Bad Ticket”}\}, \emptyset, \{guard(a_3, false)\})$$

Fig. 2. $O \equiv$ `Infoprovider` orchestration using ADORE formalism

Two special activities $\mathbb{P}$ and $\mathbb{S}$ refers to targeted orchestrations where they represent *hook* predecessors (resp. successors) in targeted partial ordering. Considering *Evolution* as a superset of *Orchestration* means that an orchestration cannot contains any *hook*, $\mathbb{P}$ or $\mathbb{S}$ occurrence. Fig. 3 represent graphically an evolution.

Substitution: Stickel defines a substitution σ in [9] as "a set of substitution components with distinct first elements, that is, distinct variables being substituted for". A substitution component is an ordered pair of two variables x and y (written as $x \rightarrow y$), denoting the replacement of the x by y (x cannot be a constant). Applying a substitution on an activity performs those replacements.

As boxes and arrows are more readable than huge sets of equations, we define a graphical syntax to represent ADORE entities. Inspired from the UML activity diagram formalism, each activity is represented as a box, and precedence rules using arrows between boxes ($a_1 \rightarrow a_2 \equiv a_1 \prec a_2$). Guards are represented as *labels* on arrows. To represent *hook* predecessors and successors, we use the *start*/*end* syntax from UML: $\bullet$ refers to $\mathbb{P}$, and $\odot$ to $\mathbb{S}$.

3.2 Merging process

Using the ADORE model, we define a merging process enabling the automatic integration of n evolutions into an orchestration. This process composes evolutions between each others, and then, merge the resulting evolution with the targeted orchestration.

Global overview: $Merge(\{e_1, \ldots, e_n\}, \{k_1, \ldots, k_m\}, o, b) \rightarrow o'$
The process takes as input a set of evolutions $\{e_1, \ldots, e_n\}$. As merge conflicts can occur, user can express some *knowledge* $\{k_1, \ldots, k_m\}$ to solve them. A knowledge k_i can be a substitution component σ_i, or new elements to add (activity, precedence rule or guard). The target of the evolution is an orchestration o and a binding $b \equiv bind(hook \rightarrow a_i)$ expresses where the evolution will be integrated inside o activities. The process results in a new orchestration o'. It does not have any side effects on existing orchestration or evolutions, as it works on a duplicated set of elements[2]. We consider it as a four steps process, described here.

1) $Merge(\{e_1, \ldots, e_n\}, \{k_1, \ldots, k_m\}) \rightarrow e'$ *("Evolution merge")*
As the global process does not have any side effect, all elements (activity, precedence) of $\{e_1, \ldots, e_n\}$ are duplicated before doing anything. This step produces a new evolution e', where all evolutions $\{e_1, \ldots, e_n\}$ are merged. From all hook points $\{h_1, \ldots, h_n\}$, we generate a new activity h' where input variables set is the union of h_i input variables set. The output variable is unified into a new

[2] Duplication implies variable renaming to avoid conflicts and new *uid*s for concerned activities.

one (using a σ). The partial ordering of e' is an union of existing partial ordering, taking care of hooks unification. Guards on h' are composed as a union of existing guards on h_i, and propagated to h' successors.

2) $DetectConflict(e') \rightarrow \{conflict_1, \ldots, conflict_n\}$ *("Conflict detection")*
At this step, we analyze the output of the previous step and detect if needed some merge conflicts[3]. Some constructions are not allowed in orchestration formalism, like *(i)*concurrent write access to a variable, *(ii)* multiple reply activities (under non–exclusive conditions), *(iii)* multiple throw activities (under non–exclusive conditions) or *(iv)* write access to a constant. To perform conflict resolution, we use an incremental approach [11]: the process automatically returns conflicts to the user, and she gives in response a knowledge (k_i, *e.g.* adding an order between two concurrent write access to a variable) to solve this conflict. The merging process is then recalled with this new knowledge.

3) $Merge(o, e', b, \{k_1, \ldots, k_m\}) \rightarrow o'$ *("Orchestration merge")*
Here, we consider that e' represents the merged evolution (output of step 2) without any conflicts. We now integrate e' into the original orchestration (following the binding specification $b \equiv \sigma(hook \rightarrow a_h)$) and produce a new orchestration o'. $\mathbb{P}$ and $\mathbb{S}$ are substituted with a_h predecessors and successors. We compute a unifying substitution where each *hook* variable (input, output) is bound to its equivalent in a_h. As in step 1, we perform guards union and propagate resulting guard set to a_h successors.

4) $DetectConflict(o') \rightarrow \{conflict_1, \ldots, conflict_n\}$ *("Conflict detection")*
This step is similar to step two except that a new kind of conflict based on evolution variables can be detected: all used variables must be declared and assigned before attempting to be read. Orchestration parameters are assigned as `receive` inputs (receiving a constant is considered as a conflict). Other variables are assigned when used as output of an activity.

As a conclusion, we can see that step 1 and 2 can be done in an abstract manner: they consist of composing evolutions, without any knowledge about target. The last steps imply real knowledge about the target, and business–specific skills. These two operations can be performed by two different users.

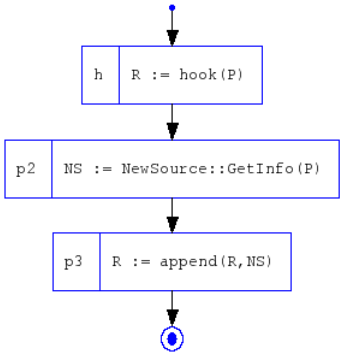
4 Illustrating merge process

This section illustrates the previously defined merge process. The first part explains on an example how activities duplication and unification works. The second part illustrates a full merge process, integrating three evolutions into the `InfoProvider` orchestration.

[3] More information about conflict detection can be found in [10].

4.1 Activities management: duplication, substitution & unification

Guided by the SEDUITE description, we can imagine a frequent action for the administrator: "How to add a new source of information ?". To automate this action, an evolution E_s is expressed (Fig. 3). E_s invokes a `NewSource` service, and appends its result to the result of the *hook* activity.



$$
\begin{aligned}
E_S \equiv\ & (\{h, p_2, p_3\},\\
& \{\mathbb{P} \prec h, h \prec p_2, p_2 \prec p_3, p_3 \prec \mathbb{S}\})\\
h \equiv\ & (h, hook, \{P\}, R, \emptyset)\\
p_2 \equiv\ & (p_2, invoke(NewSource, GetInfo),\\
& \{P\}, NS, \emptyset)\\
p_3 \equiv\ & (p_3, assign(append), \{R, NS\}, R, \emptyset)
\end{aligned}
$$

Fig. 3. $E_s \equiv$ "How to add a new source of informations ?"

E_s describes *abstractly* how to add a new source of informations. If an administrator wants to add a `Weather` source of information, she will specialize the semantic of this evolution using a substitution σ_x. Now, we consider two evolutions obtained by substitutions from the previous one (E_w adding a `Weather` source, and E_e adding an `Events` source) and focus on the first and second steps of the merge process, *i.e.* applying $E_{w \oplus e} \equiv Merge(\{E_w, E_e\}, \emptyset)$.

$$
\begin{aligned}
\sigma_w \equiv \sigma(\{NewSource \rightarrow Weather\}) \Rightarrow E_w \equiv \sigma_w(E_s)\\
\sigma_e \equiv \sigma(\{NewSource \rightarrow Events\}) \Rightarrow E_e \equiv \sigma_e(E_s)
\end{aligned}
$$

First of all, we duplicate each activity to avoid naming conflict (*uids*, *variable* names). The duplication produces the following result:

$$
\begin{aligned}
E_{E_w \cup E_e} \equiv\ & rename(E_w) \cup rename(E_e)\\
\equiv\ & (\{h^w, p_2^w, p_3^w, h^e, p_2^e, p_3^e\},\\
& \{\mathbb{P} \prec h^w, h^w \prec p_2^w, p_2^w \prec p_3^w, p_3^w \prec \mathbb{S}, \mathbb{P} \prec h^e, h^e \prec p_2^e, p_2^e \prec p_3^e, p_3^e \prec \mathbb{S}\})\\
h^w \equiv\ & (h^w, hook, \{P_w\}, R_w, \emptyset)\\
p_2^w \equiv\ & (p_2^w, invoke(Weather, GetInfo), \{P_w\}, NS_w, \emptyset)\\
p_3^w \equiv\ & (p_3^w, assign(append), \{R_w, NS_w\}, R_W, \emptyset)\\
h^e \equiv\ & (h^e, hook, \{P_e\}, R_e, \emptyset)
\end{aligned}
$$

$$p_2^e \equiv (p_2^e, invoke(Events, GetInfo), \{P_e\}, NS_e, \emptyset)$$
$$p_3^e \equiv (p_3^e, assign(append), \{R_e, NS_e\},\ R_e, \emptyset)$$

The process will now perform merge of h^e and h^w into h. As R_w and R_e are output variables of a unified activity, they must be unified too in a R variable. The process does not have any knowledge k_i about inputs parameters P_w and P_e, and treat them as two different variables. This merge produces a substitution we apply onto the merged evolution to produce Fig. 4 result. The conflict detection step ($DetectConflict(E_{w\oplus e})$) will return a conflict, as there is no precedence rule between two different write into variable R.

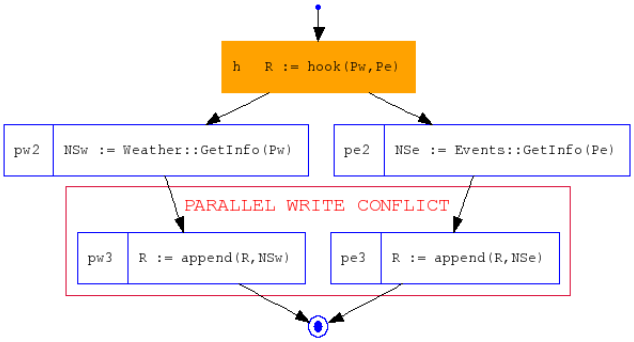


Fig. 4. $Merge(\{E_w, E_e\}, \emptyset) \Rightarrow ConcurrentWriteConflict(R, \{p_3^w, p_3^e\})$

To solve this conflict, we add a precedence rule between these two activities. If we consider that `Weather` information is more important than `Events` one, we can express a knowledge k_1 to represent it. Even if there is no conflict leading to it, we can also specify to the merging process using k_2 that the *hook* input variables have to be substituted to the same P variable.

$$k_1 \equiv augmentOrder(p_3^w \prec p_3^e)$$
$$k_2 \equiv \sigma(\{P_w \rightarrow P, P_e \rightarrow P\})$$

These knowledge allow the merge process to perform full merge, and conflict detection returns an empty set of conflicts. We obtain as output a new evolution $E_{w\oplus e}$, Fig. 5.

4.2 Practicing the merge algorithm

In this section, we consider that the administrator wants to perform the following evolutions inside `InfoProvider` (Fig. 2): *(i)* add a weather source of information (E_w), *(ii)* add an events source of information (E_e) and *(iii)* check profile

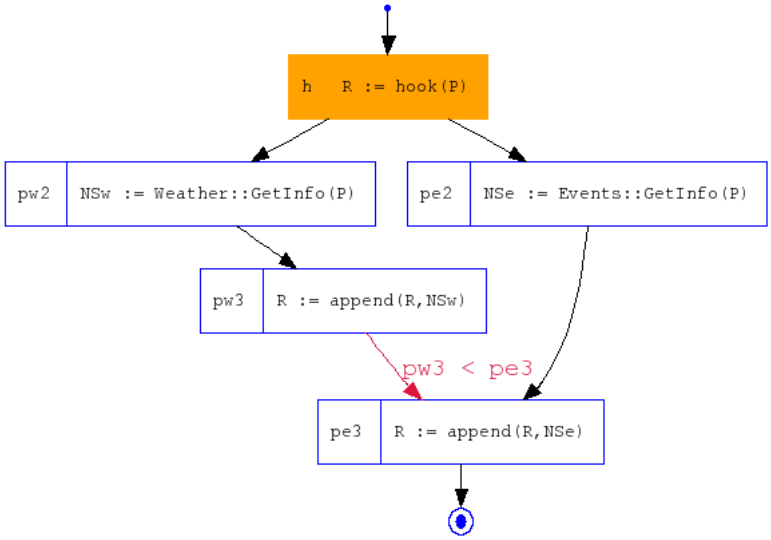


Fig. 5. $E_{w\oplus e} \equiv Merge(\{E_w, E_e\}, \{k_1, k_2\})$

correctness before attempting to retrieve informations (E_p). E_p asks a service to verify the given profile, and throws an exception if this profile is not correct (Fig. 6).

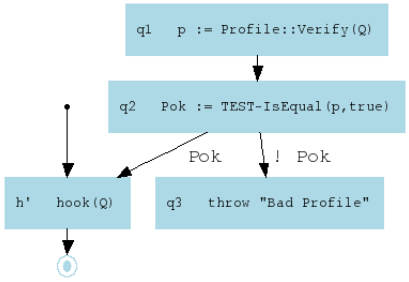


Fig. 6. $E_p \equiv$ Checking profile correctness evolution

As we attempt to merge E_w and E_p for a second time, we reuse the knowledge set $\{k_1, k_2\}$ from previous section and avoid the conflict detection step to clarify text. The administrator knows that all hook parameters should be unified, as they all refer to `profile`. She expresses this knowledge by adding a knowledge

k_3 expressing this unification between hook parameters:

$$k_1 \equiv augmentOrder(p_3^w \prec p_3^e)$$
$$k_2 \equiv \sigma(\{P_w \rightarrow P, P_e \rightarrow P\})$$
$$k_3 \equiv \sigma(\{Q \rightarrow P\})$$

Following the merge algorithm, we perform $Merge(\{E_w, E_e, E_p\}, \{k_1, k_2, k_3\})$ and compute $E_{w \oplus e \oplus p}$ (Fig. 7) as a result of the first merge step. The guard on *hook* coming from E_p is propagated to successors of hook (now guarded by $guard(q_2, true)$). As there is no conflict detected at second step, we can perform the orchestration merge. Our evolution must be hooked on information retrieving activity in o, *i.e.* a_4.

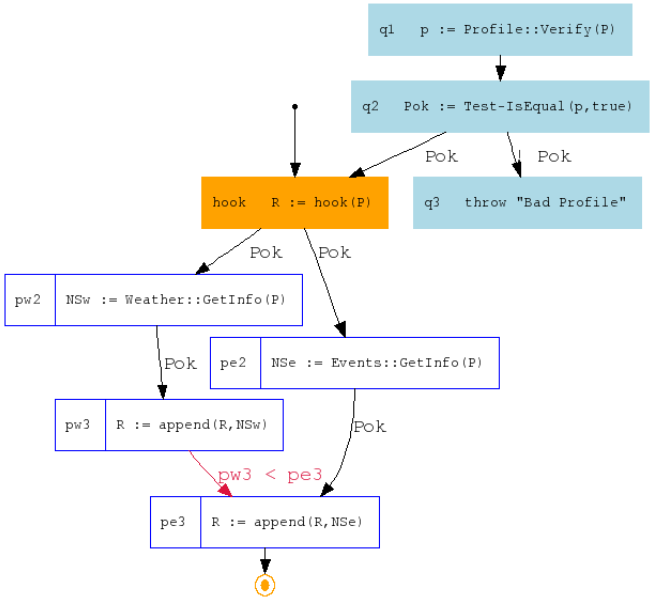


Fig. 7. $E_{w \oplus e \oplus p} \equiv Merge(\{E_w, E_e, E_p\}, \{k_1, k_2, k_3\})$

When invoking $Merge(o, E_{w \oplus e \oplus p}, bind(hook \rightarrow a_4)\{k_1, k_2, k_3\})$ at third step, the merge process binds *hook* with a_4. We perform usual substitution of *hook* output variable with a_4 output variable, without any conflict. Moreover, as a_4 and *hook* activities have only one input variable, we can deduce a unification between these two variables. The activity q_1 interacts with a_1, and it generates an *UnassignedVariable* conflict (q_1 read `profile` content, but has no predecessors). As `profile` is an input of o, this assignment is performed by the `receive` activity a_1. We can automatically add a precedence rule $a_1 \prec q_1$ to solve this conflict. Fig. 8 represents the final orchestration o', result of the merge process.

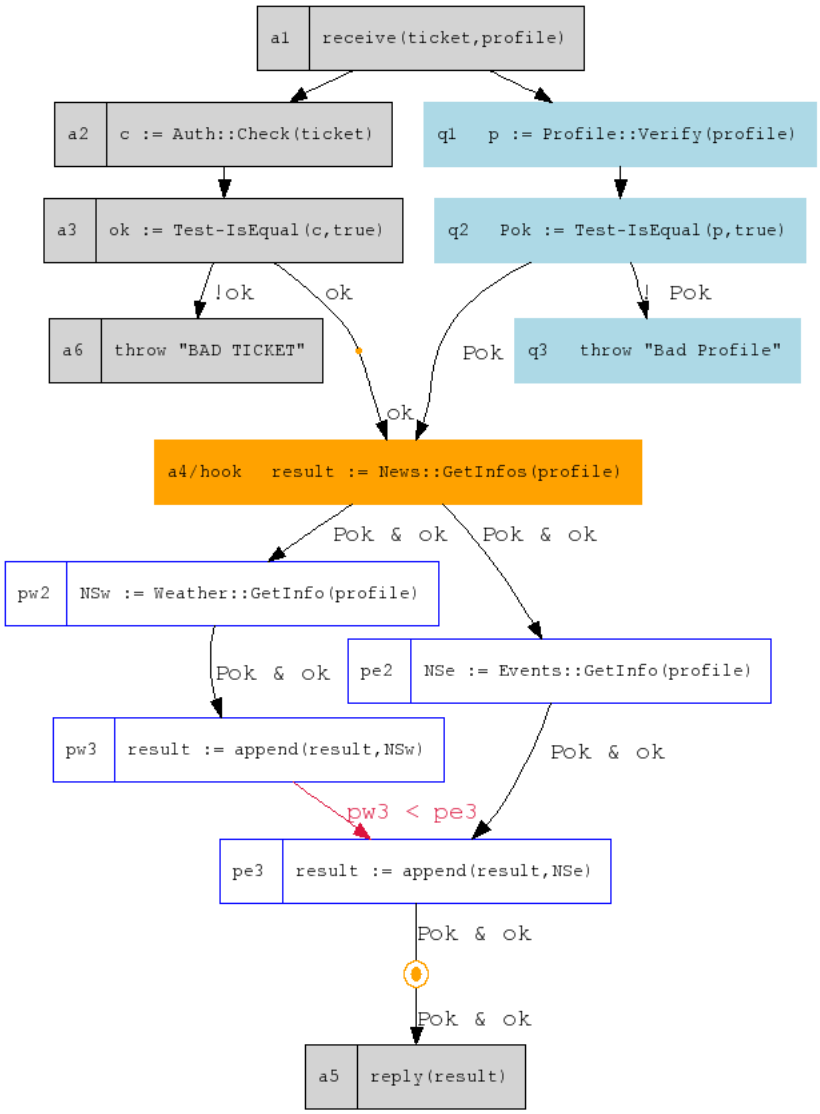


Fig. 8. $O' \equiv Merge(\{E_w, E_e, E_p\}, \{k_1, k_2, k_3\}, o, \sigma(hook \rightarrow a_4))$

5 Validation & Implementation

SEDUITE *example analysis:* We can analyze the example shown previously using some metrics. We compare the obtained behavior with respect to three parameters: *(i)* $|Act|$ the cardinality of activities set, *(ii)* $|\prec|$ the cardinality of partial orderings and *(iii)* $|k|$ the cardinality of the knowledge set. We use the $|X_m|$ notation to express how many elements of $|X|$ were impacted by the merge process
.

Tab 2 exposes results of this analysis. The last line analyzes the final orchestration o'. We can see that 93% of precedence rules can be automatically managed by the merge process. Moreover, 54% of activities must be adapted to be consistent with the evolution, and all these adaptations are automatically performed by the merge process.

Behavior	$\|Act\|$	$\|Act_m\|$	$\|\prec\|$	$\|\prec_m\|$	$\|k\|$	$\|k_m\|$
$E_{w\oplus e}$	5	5 (100%)	7	6 (85%)	2	–
$E_{p\oplus w\oplus e}$	8	6 (75%)	10	9 (90%)	3	–
O'	13	7 (54%)	14	13 (93%)	4	1 (25%)

Table 2. Measuring *Merge* impact on SEDUITE example

Implementation: the merge process is implemented using PROLOG. It allows partially–automated orchestration evolution. Based on EMF [12], we develop a model-driven software to deal with the merge process. A JAVA object front–end allows final user to interactively use the PROLOG merge engine in a user–friendly way. More information can be found on ADORE web-site[4].

The SEDUITE system is used to support research on user profiles management [13] and to validate a French national research project called FAROS[5], dealing with SOA reliability. The `InfoProvider` orchestration is implemented using the BPEL 2.0 standard. It runs over the APACHE ODE open–source orchestration engine. Seduite applications should be deployed in different academic institutions. We are working on a user-friendly environnment that supports controled evolutions. It 's the current validation for web services orchestrations evolution merging. More information about SEDUITE implementation can be found on SEDUITE web site[6]. Further validation is an ongoing work. We focus our experiments on large–scale work-flows from grid–computing research field.

6 Related work & Discussions

In [14], Rémy Douence defines *Aspect Oriented Programming* (AOP) as:

[4] `http://rainbow.i3s.unice.fr/adore`
[5] `http://www.lifl.fr/faros` (french only)
[6] `http//anubis.polytech.unice.fr/jSeduite`

> AOP *is a set of language mechanisms which enable the introduction of non anticipated functionalities in a base application. Without these mechanisms, the code of the base application should be modified in several locations.* AOP *enables to modularise these functionalities*

Existent work [15, 16] bind AOP concepts to Orchestrations. These approaches weave aspects inside the orchestration engine, and allow integration of unforeseen evolutions directly into the targeted orchestration. These approaches imply modifications of orchestration engines to add the aspect weaver inside it. Moreover, users will have to *(i)* implement their orchestrations using BPEL, *(ii)* use the aspect language to express new functionality and finally *(iii)* deploy orchestrations and weave aspects into an *ad'hoc* engine. Aspects interactions (*i.e.* different aspects woven at the same point) are solved by ordering aspect codes into block (this code will be executed before that one). AOP considers special keywords to decorate *pointcuts* with *advices* (*before*, *around*, *after*) and use `proceed` keyword to represent normal behavior of woven code. Considering the *hook* as a `proceed` and its binding as a *pointcut* selection, our approach supports the enhancement of the original behavior of an orchestration. As we reify and then merge behavior instead of expressing advice as black boxes and ordering them, we focus on parallel execution of distinct evolutions (we do not create order between evolutions unless it is necessary). Moreover, the ADORE behavior reification allows evolution to be composed in a quasi–automatic way, detecting conflits to ensure orchestration validity. Contrarily to AOP which can be dynamic, our approach only considers static evolutions. The evolution process is done at model level and then refined after success into BPEL code using usual models transformation tools. This approach does not require any specific orchestration engine. As soon as orchestration engines will support dynamic orchestration changes, our approach could be used at runtime.

Former work [17] defines an associative and commutative merge algorithm dealing with components interactions. As this approach is a superset of the previous one focused on orchestrations and WSOA, we can restrict the algorithm described here to ensure associativity and commutativity. If we consider evolutions always using a usual schema[7] and no conflict resolution rules, we can ensure theses properties. Proving associativity and commutativity in others cases is an ongoing work.

Similar to the UML templates, evolutions define generic orchestration view whose some variables (template parameters) need to be bound. The ADORE model is dedicated to orchestration merging and binding step implies automatic composition of elements. Knowldege is used to reduce the space adressed by an evolution or to solve a conflict during this automatic merging. It plays the equivalent role as composition directives as defined in [18] to compose models.

In conclusion, we claim that a high level reasoning model such as our merge algorithm helps composition mechanisms.

[7] $\mathbb{P} \prec hook \prec \mathbb{S}$

7 Conclusions & Perspectives

In this article, we addressed the problem of Web Services orchestrations evolution on behavioral part. After identifying some needs of evolution capability on a real case orchestration, we restrict our investigations to behavioral evolutions. A formal model called ADORE is proposed and we define a merge algorithm able to compose in a quasi–automatic way evolutions into orchestrations. We also show on an example how the algorithm works, detailing each steps of the merge process and focusing on non–trivial parts. Furthermore, we show how this process can help evolution composition and discuss our work with respect to related work.

The work presented in this paper collaborates with the WSOA administrator for resolving semantic choices. She will help variable unification and conflict resolution using her knowledge and skills on the system. As the knowledge domain is closed to enterprise system boundaries, we can imagine some semantic web mechanisms to express such knowledge and capitalize enterprise knowledge.

We only consider evolution processes as an enrichment of an orchestration. We never address the activities removal problem or activity substitution. AOP define a *delegate* keyword to substitute the original behavior with a new one [19]. The merge algorithm will have to be modified to take care of this kind of evolution. So far, we consider evolution removal as a very simple operation: we take the original orchestration and the set of wanted evolutions, excepted the removed one. Finer grain mechanisms can be envisaged to deal with evolution removal.

The merge algorithm presented here considers only one *hook* binding inside the original orchestration. But it could be useful to allow multiple bindings (*e.g.* selecting all `reply` activities inside an orchestration) at merge time. We also consider that a *hook* point is an atomic activity. But evolutions can be applied to a block of activities, adding scope concerns to the merge process [20]. These two considerations implies the composition of overlapping evolutions when such a situation occurs.

References

1. MacKenzie, M., Laskey, K., McCabe, F., Brown, P., Metz, R.: Reference Model for Service Oriented Architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS (February 2006)
2. Peltz, C.: Web services orchestration and choreography. Computer **36**(10) (2003)
3. W3C: Web service glossary. Technical report (2004)
4. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution, Washington, DC, USA, IEEE Computer Society (2005) 13–22
5. Lehman, M.M.: Laws of software evolution revisited. In: EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology, London, UK, Springer-Verlag (1996) 108–124

6. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
7. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a taxonomy of software change. Journal on Software Maintenance and Evolution: Research and Practice **17**(5) (September-October 2005) 309–332
8. Jordan, D., Evedmon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, K., Khalaf, R., Konig, D., Marin, M., Mehta, V., Thatte, S., Van der Rijn, D., Yendluri, P., Yiu, A.: Web services business process execution language version 2.0. Technical report, OASIS (2007)
9. Stickel, M.E.: A unification algorithm for associative-commutative functions. J. ACM **28**(3) (1981) 423–434
10. Nemo, C., Blay-Fornarino, M., Kniesel, G., Riveill, M.: SEMANTIC ORCHESTRATIONS MERGING - Towards Composition of Overlapping Orchestrations. In Filipe, J., ed.: 9th International Conference on Enterprise Information Systems (ICEIS'2007), Funchal, Madeira (June 2007)
11. Mens, T., Van Der Straeten, R.: Incremental resolution of model inconsistencies. In: WADT 2006. Volume 4409 of Lecture Notes in Computer Science., Springer-Verlag (2007) 111–126
12. Merks, E., Eliersick, R., Grose, T., Budinsky, F., Steinberg, D.: The Eclipse Modeling Framework. Addison Wesley (2003)
13. Joffroy, C., Pinna-Déry, A.M., Renevier, P., Riveill, M.: ARCHITECTURE MODEL FOR PERSONALIZING INTERACTIVE SERVICE-ORIENTED APPLICATION. In: 11th IASTED International Conference on Software Engineering and Applications (SEA'07), Cambridge, Massachusetts, USA, IASTED, ACTA Press (November 2007) 379–384
14. Douence, R.: A restricted definition of AOP. In Gybels, K., Hanenberg, S., Herrmann, S., Wloka, J., eds.: European Interactive Workshop on Aspects in Software (EIWAS). (September 2004)
15. Charfi, A., Mezini, M.: Aspect-oriented web service composition with ao4bpel. In: ECOWS. Volume 3250 of LNCS., Springer (2004) 168–182
16. Courbis, C., Finkelstein, A.: Weaving aspects into web service orchestrations. In: ICWS, IEEE Computer Society (2005) 219–226
17. Blay-Fornarino, M., Charfi, A., Emsellem, D., Pinna-Déry, A.M., Riveill, M.: Software interaction. Journal of Object Technology (ETH Zurich) **3**(10) (2004) 161–180
18. Reddy, Y.R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., McEachen, N., Song, E., Georg, G.: Directives for composing aspect-oriented design class models. **3880** (2006) 75–105
19. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning (July 2003) ISBN-10: 1930110936 ISBN-13: 978-1930110939.
20. Klein, J., Fleurey, F., Jézéquel, J.M.: Weaving multiple aspects in sequence diagrams. Transactions on Aspect-Oriented Software Development (TAOSD) **LNCS 4620** (2007) 167–199