Dealing with Changes in Service Orchestrations

Leandro Sales Pinto
Politenico di Milano
Dipartimento di Elettronica e Informazione - DEI
Piazza Leonardo Da Vinci, 32
Milano, Italy
pinto@elet.polimi.it

Gianpaolo Cugola
Politecnico di Milano
Dipartimento di Elettronica e Informazione - DEI
Piazza Leonardo Da Vinci, 32
Milano, Italy
cugola@elet.polimi.it

Carlo Ghezzi
Politecnico di Milano
Dipartimento di Elettronica e Informazione - DEI
Piazza Leonardo Da Vinci, 32
Milano, Italy
ghezzi@elet.polimi.it

ABSTRACT

Service Oriented Computing (SOC) allows programmers to build distributed applications by putting together (i.e., *orchestrating*) existing services exported by remote providers. The main source of complexity in building such kind of orchestrations is the need for anticipating and explicitly handling (i.e., programming ad-hoc countermeasures) possible changes in the external environment that may affect them, like faults invoking services removed by their providers.

To ease the job of programmers we developed DSOL, an innovative infrastructure supporting design and execution of service orchestrations. It combines a declarative approach to model the orchestration with planning mechanisms to actually run it. In this paper we focus on the mechanisms provided by DSOL and its associated execution engine to deal with changes that may happen at runtime. In particular, we show how the declarative nature, the modularity, and the dynamism inherent in the DSOL approach allows changes to be easily managed, both those that were forecasted at design time and those that require the workflow to be changed while the orchestration is running.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Service oriented programming, self-adaptation

Keywords

Service orchestration, changes management, exception handling

1. INTRODUCTION

Service orchestrations live in a very unstable world in which changes occur continuously and unpredictably. External services invoked by the orchestration may be discontinued by their providers, they may fail, or they may become unreachable or incompatible with the original versions. Furthermore, the orchestration requirements may evolve due to business needs. It is therefore fundamental that orchestration languages and their run-time systems provide ways to support dynamic evolution, allowing orchestration models or even currently running instances to be modified to cope with unforeseen situations and changes in requirements.

To better illustrate these needs, we introduce a case study, which we will also use throughout the paper to describe our approach. It is a web service used by a new bookstore to perform its sales. Initially, this service composes the following operations: an internal service that checks if the desired book is available in stock, an external service that handles the payments, and another external service used to contact the business partner that handles deliveries. In particular, let us consider the following requirements:

- At service invocation, the client provides the relevant information about the books she wants to buy, the delivery address and the details about payment;
- First action to perform is checking the availability of books. If they are in stock the order is saved with status *open*. Otherwise, an exception is thrown;
- After checking availability and saving the order, the process continues to the payment stage. For payment, two alternatives are available: PayPal or credit card. The preferred option is to use PayPal, because in such way the bookstore does not need to receive or keep any information about payments, e.g., credit card numbers, limiting its responsibilities. PayPal itself also offers much more alternatives of payments to the client. If, for any reason, the system is not able to invoke the PayPal services, it enables the use of credit card with two alternatives. The first is to contact the card operator directly, using its services API. In case this route fails, the last alternative is to save the credit card information so that the payment can be done manually. In such case, the status of the order becomes *payment pendent*, (otherwise it is *payment authorized*). If the payment is not authorized, an exception is thrown;
- The delivery can only be scheduled after the payment. Depending on the payment status, the delivery is scheduled as *immediate* or *wait for confirmation*.

As a new bookstore that wants to enter the market gradually, due to contractual costs reason, the managers decided

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

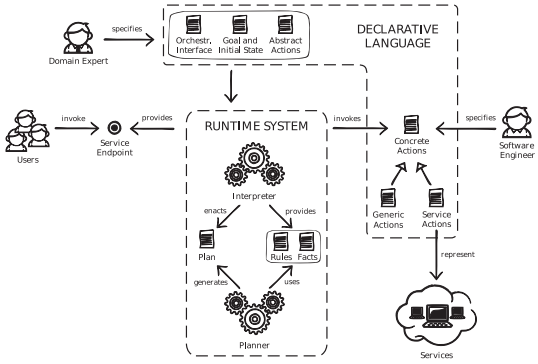

Figure 1: The DSOL approach

to initially sell only to national destinations. However, during this initial period the bookstore receives a huge, international order request from a foreign university. Although it is still not worth to start selling worldwide, the manager sees this situation as an interesting business opportunity and wants to fulfill this request. The problem is that neither the current system is able to handle it, because international orders require an extra insurance in case of miscarriage, nor the current deliver company is able to handle international deliveries. In order to complete this order, the system needs to deviate [11] from the current process, including the mandatory insurance, and also contacting a delivery company able to deliver at the required destination.

Another natural evolution would be to accept international orders and ship worldwide. The main difference of this case w.r.t. the previous one is that the former is a deviation that applies only to a specific running instance of the process, while the latter is an evolution of the whole process, which will affect all further executions.

To support dynamic evolution of service orchestrations, we developed *DEng – DSOL Execution Engine*[1], an engine based on the declarative language *DSOL – Declarative Service Orchestration Language*. In [12] we provided a general description of DSOL and its declarative nature, here we focus on changes and how they are supported by our system.

In particular, the next section describes how the declarative and dynamic approach to orchestration modeling and execution offered by our system greatly simplify the job of developing flexible orchestrations that could efficiently manage various types of *foreseen* deviations from the standard course of actions. In Section 3 we instead focus on the set of mechanisms we recently added to DEng in order to cope with *unforeseen* changes, by allowing the orchestration model/instances to evolve at runtime. Finally, a comparison with related works is given in Section 4, while Section 5 draws some conclusion and discusses future work.

2. BUILDING FLEXIBLE ORCHESTRATIONS WITH DSOL

Using traditional languages, like BPEL [3] and BPMN [29], a service orchestration is modeled as a monolithic program, which captures the entire flow of execution, from the start of the orchestration to the invocation of the elementary services in charge of executing each step. Changes in the external environment that force to deviate from the expected course of actions are usually dealt with by heavily using exception handling and compensation mechanisms intertwined with the process code; an approach that makes orchestrations hard to write, understand, and modify.

DEng adopts a radically different approach. A service orchestration is modeled in terms of its goals and the actions to reach them, using the declarative language DSOL. The DEng runtime system then uses planning techniques to determine the actual flow of execution to achieve the orchestration's goals (i.e., which actions to execute and in which order). The same planning techniques are also used in case of faults, to build alternative paths of execution without the need for explicitly programming them.

This eases the job of building flexible orchestrations capable of coping with faults and changes in the external environment, while the modularity and dynamism inherent in such approach also simplifies the steps required to change the orchestration model at runtime (which is the main focus of the new features added to DEng, presented in Section 3).

The DSOL model of a service orchestration includes different aspects, which are defined separately using different idioms, possibly by different stakeholders, each bringing their own competences. In particular, as shown in Figure 1, a DSOL model includes the elements described hereafter.

Orchestration Interface

The *orchestration interface* formalizes how the orchestration is exposed as a web service and is defined as a Java interface compliant with the JAX-WS [2] specification. Listing 1 shows the orchestration interface of our reference case study.

```
@WebService
public interface Bookstore {
 @WebReturn(name='order')
 public OrderInfo order(
  @WebParam(name='books_list')
  List<Book> books,
  @WebParam(name='deliveryAddress')
  Address deliveryAddress,
  @WebParam(name='paymentDetails')
  PaymentDetails pd);
}
```

Listing 1: The bookstore orchestration interface

Orchestration Goal and Initial State

The *goal* of the orchestration, which is usually expressed by a domain expert not necessarily competent in software development, is a set of facts that represent the expected state of the world after executing the orchestration. Similarly, the *initial state* models the set of facts that one can assume to be true at orchestration's invocation time.

Note that to add flexibility the goal may actually include a *set of states*, which reflect the various alternatives to accomplish the goal of the orchestration, listed in order of preference. This is the first mechanism provided by DEng to manage possible changes in the external environment; a mechanism that does not require to explicitly handle exceptions or to code different paths of executions. Indeed, by listing different goals in order of preference, the domain expert may easily distinguish between the preferred course of actions and the available alternatives in case something goes wrong and the preferred goal cannot be reached.

[1]A prototype implementation of DEng is available for download at `http://www.dsol-lang.net`

```
start true

goal
  (booksAvailableInStock and orderSaved   and
   paymentDoneByPayPal   and deliveryScheduled)
or
  (booksAvailableInStock   and orderSaved   and
   paymentDoneByCreditCard and deliveryScheduled)
```

Listing 2: Initial state and goal

As an example, Listing 2 shows the initial state and the goal for the bookstore scenario. In particular, two alternative goals are listed, the preferred one that describes the situation when payment has been done through PayPal, and the alternative to follow if paying through PayPal is impossible. As for the initial state, to model the bookstore scenario we do not need to assert any special fact, so the initial state becomes *true*. Notice however that the DEng runtime system automatically populates the initial state with facts that reflect the orchestration arguments: *books_list(books)*, *address(deliveryAddress)*, and *paymentDetails(pd)*.

Abstract Actions

Abstract actions are high-level descriptions of the primitive actions available in a given domain, which DEng uses as the building blocks of orchestration plans. They are modeled in an easy-to-use, logic-like language, in terms of their *signature*, *precondition*, and *postcondition*. We expect that such language can be mastered by non-technical domain experts.

```
action checkStock(Books)
pre:  books_list(Books)
post: booksAvailableInStock

action saveOrder(Books)
pre:  books_list(Books), booksAvailableInStock
post: orderSaved, order_info(order)

action payByPayPal(Order,PD)
pre:  order_info(Order), paymentDetails(PD)
post: paymentDoneByPayPal

action payByCreditCard(Order,PD)
pre:  order_info(Order),paymentDetails(PD)
post: paymentDoneByCreditCard

action scheduleDelivery(Order, Address)
pre:  order_info(Order), deliveryAddress(Address)
post: deliveryScheduled
```

Listing 3: Abstract actions

Note that while writing the abstract actions, the domain expert may focus on the general aspects of the domain, leaving out all the implementation details including the expected sequence of execution. As an example, Listing 3 illustrates the abstract actions that model the bookstore scenario. On one hand, we notice that the fact that the payment by credit card can be implemented in two different ways (automatically or manually) is not visible at this level. Similarly, we notice that there is no reference to the order of execution, which will be automatically chosen by the Interpreter (see Figure 2) at runtime in order to satisfy the goal.

This approach of focusing on the general actions available in a domain, leaving the actual workflow to be automatically decided at runtime, is the second mechanism provided by DEng to help modelers in defining a flexible orchestration that may easily cope with changes.

Concrete Actions

Concrete actions are the executable counterpart of abstract actions and represent the concrete steps required to implement the operations modeled by the abstract action, e.g., by invoking an external service or executing some code. They are meant to be specified by a different stakeholder, i.e., a software engineer with programming skills, once the abstract actions have been identified and specified by the domain expert. Concrete actions are implemented through Java methods using an ad-hoc annotation *@Action* to refer to the abstract actions they implement.

```
@Action(name="payByPayPal",service="paypal")
public abstract void payByPayPal(OrderInfo order,
                                 PaymentDetails pd);

@Action(name="payByCreditCard",service="visa")
public abstract void payByCreditCard(OrderInfo order,
                                     PaymentDetails pd);

@Action(name="payByCreditCard")
public void savePaymentInfo(OrderInfo order,
                            PaymentDetails pd){
        order.setStatus(PAYMENT_PENDENT);
        order.setPaymentDetails(pd);
        order.save();
}
```

Listing 4: Example of concrete actions

Among concrete actions, we distinguish between *service actions* and *generic actions*. Service actions are abstract Java methods directly mapped to external services. As an example see the method *payByPayPal* in Listing 4. Service actions have a special attribute `service` of the *@Action* annotation that specifies the external service to invoke, while a hot-pluggable module of the DEng runtime system is responsible for taking this information and finding the specified service to be invoked. This way, service actions may represent different kinds of services, e.g., SOAP or RESTful. DEng can be easily extended to support other SOA technologies. Generic actions are simple Java methods that can perform any operation. As an example, see the method *savePaymentInfo* in Listing 4. It changes the state of the order and saves the payment information into the database.

It is important to note that, in general, several concrete actions may be bound to the same abstract action. This way if the currently bound concrete action fails, i.e., it returns an exception, the Interpreter has other options to accomplish the orchestration step specified by the failed abstract action. As an example, Listing 4 illustrates two different alternatives to accomplish the abstract action *payByCreditCard*. This ability to associate several concrete actions to the same abstract action (the choice of which one to use being left to the runtime system) is the third mechanism implemented in DEng to easily build flexible orchestrations, capable of handling changes in the external environment without the need of explicitly programming how to manage them.

2.1 Execution

At orchestration invocation time (see Figure 2) the *Interpreter* translates the goal, the initial state, and the abstract actions into a set of *rules* and *facts* used by the *Planner* to build an abstract plan of execution, which lists the logical steps through which the desired goal may be reached. Listing 5 illustrates a plan for our example. It includes a list of abstract actions that can lead from the initial state to a

state that satisfies the first orchestration goal.

```
checkStock(books)
saveOrder(books)
payByPayPal(order,pd)
scheduleDelivery(order,deliveryAddress)
```

Listing 5: A possible plan for the bookstore example

This plan is taken back by the Interpreter, which enacts it by associating each step (i.e., each *abstract action*) with a *concrete action* that is executed, possibly invoking external services. It is important to note that while the plan is described as a sequence of actions, the Interpreter is free to execute them in parallel, by invoking each of them as soon as their precondition is satisfied.

If something goes wrong during execution (e.g., an external service is unable to accomplish the expected task or it returns an exception), the Interpreter first tries a different concrete action for the abstract action that failed. If this is not enough, the Interpreter invokes the Planner again to find a different course of actions that could skip the step that failed. For example, if the action *payByPayPal* fails the Planner is invoked and it computes another plan that uses the action *payByCreditCard* instead of the faulty one. Furthermore, by comparing the old and the new plan, considering the current state of execution, the Interpreter is able to calculate the set of actions that need to be *compensated* (i.e., undone) as they have already been executed but are not part of the new plan. Compensation actions are defined following the same idea of concrete actions. In such case, the attribute *compensate* is added to annotation *@Action*. An example of compensating action is shown in Listing 6, which shows the steps that are executed if the action *saveOrder* need to be undone. Note that we do not expect that all action could be undone. We just provide mechanisms to accomplish this step when possible.

```
@Action(name="saveOrder", compensation=true)
public void cancelOrder(@ObjectName("order")
                        OrderInfo order){
        order.setStatus(CANCELED);
        order.save();
}
```

Listing 6: Save order compensation action

This plan-execute-replan process is repeated until one plan is found that successfully reaches one of the orchestration goals or a plan cannot be built. In the first case, a successful message is sent to the client. Otherwise, the last tried plan is compensated and an exception is thrown.

As mentioned at the beginning of this section, the DEng approach to orchestration modeling and execution supports workflow management that can handle quite nicely adverse behaviors in the external environments, which may otherwise lead to failures, or whose treatment in a traditional orchestration language (like BPEL) would lead to difficult to write and understand code.

Taking our case study as an example, the mechanisms described so far allow DEng to handle the case where the PayPal service cannot be accessed (by anticipating this situation and including two abstract actions for payment) and the case where the credit card cannot be charged on-line (by defining two different concrete actions for the same abstract action *payByCreditCard*). The aspect that is yet uncovered is support to handling unforeseen changes, i.e., those that need the orchestration model to be modified at runtime, like handling an unexpected international order request. This is the topic of the next section.

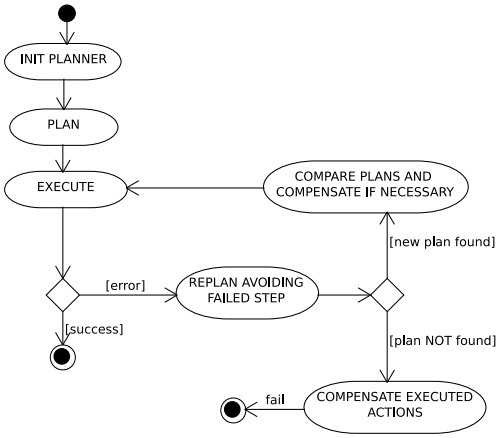

Figure 2: Process followed by the Interpreter to execute a service orchestration

3. CHANGES AT RUNTIME

While the declarative nature of DSOL allows easily modeling of flexible orchestrations, the modularity and dynamism inherent in the DEng approach provide a perfect substrate where ad-hoc mechanisms can be added to change the orchestration at runtime. Indeed, as the plan of execution, i.e., the actual sequence of activities to perform, is built at runtime, changing the orchestration is much simpler in DEng compared to the complex mechanisms that other, more traditional systems must put in place to obtain the same result.

In particular, as we explain in detail in the remainder of this section, changing the orchestration at runtime requires the plan of a running orchestration to be re-built from the current state of execution: something very similar to what the Interpreter already does to bypass a faulty situation that blocks the current plan.

In general we support full changes to the orchestration model. The service architect may add new abstract or concrete actions, remove or modify them, change the goal of the orchestration, and even change its interface. Moreover, we allow changes that impact the orchestration at various levels. Indeed, when the architect submits a new model for an existing orchestration it has to specify if it has to affect future executions, current ones, or both. This way, we cover different levels of updates: from small changes, applied to single running instances, to changes to be applied to future calls only, to major changes that have to affect current and future executions.

To better characterize this aspect, we define the concept of *orchestration instance* as the running orchestration that is created each time a request is made to the interface service of the orchestration. Such instance is represented internally by an *instance descriptor*, which we extended to include a copy of the orchestration model as it was defined at the time when the orchestration was invoked.

Given this premise, we notice that the case of a change that must affect only new instances does not creates special problems. What we do is to let current instances proceed

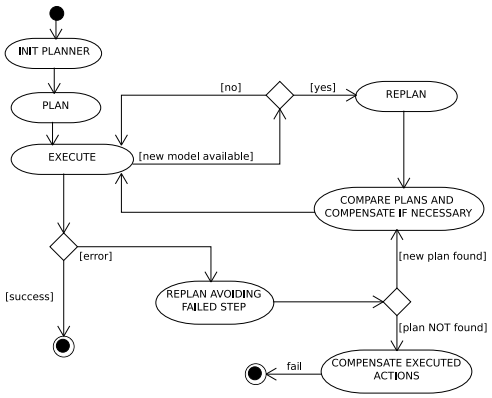

Figure 3: Process followed by Interpreter to execute an orchestration supporting changes at runtime

using their own copy of the model, while the main copy, used for future calls, is overwritten with the new one.

The situation is more complex when the new model has to be applied to running instances. Indeed, this requires modifying the way the Interpreter operates. In particular (see Figure 3), as soon as the new model is submitted, the Interpreter stops executing the current plan and invokes the Planner to build a new plan in line with the new model. At this point, as it happens for standard re-planning, the new plan is compared with the old one to decide where to start executing it and which action to undo, if any.

Notice that in applying a new model to a running instance we do not consider changes to the orchestration interface, which are taken into consideration only for future calls. Also notice that the declarative nature of DSOL and the modularity and dynamism inherent in DEng eliminate most of the typical problems about possible mismatches between the state of current executions and the changed model. In particular, during the re-planning phase that follows the submission of a new model, the Planner does not start from a generic "*true*" state, but it takes into consideration the current state of execution, i.e., the facts already asserted during the execution of the old plan. This guarantees that the new plan, if found, is coherent with the current state.

To put in evidence the potential of these mechanisms, consider our case study. When the order request from the foreign university arrives we may decide to accept it by adding to the original model the abstract actions described in Listing 7 with the related concrete actions (omitted for brevity). As we do not want this new behavior to be shared by the whole system, those changes will be applied only to the running instance that received the request. This instance, instead of failing because the *scheduleDelivery* action could not be performed toward an international delivery, would detect those changes and trigger the re-plan phase. The new plan would include the *scheduleInternationalDelivery* action, that replace the *scheduleDelivery* actions to accomplish part of the goal, and the *buyOrderInsurance* that satisfies one of the pre-conditions of *scheduleInternationalDelivery*. The new plan executes successfully and allows to manage this exceptional situation.

```
action buyOrderInsurance(Order)
pre:  order_info(order)
post: orderInsuranceDone
```

```
action scheduleInternationalDelivery(Order, Address)
pre:  orderInsuranceDone, order_info(Order),
      deliveryAddress(Address)
post: deliveryScheduled
```

Listing 7: New abstract actions for small deviation

On the other hand, imagine that at some time the managers of our bookstore had the chance to find two good partners to handle the international deliveries. The first and preferred one provides also the insurance of the order. The second one, although it deliveries to a broaden number of destinations, does not include the insurance.

To include the first partner into the job and open the bookstore to international clients, changing the concrete actions that implement the *scheduleDelivery* is enough. This is shown by Listing 8, which shows the concrete actions to include the first partner. Notice that we used the *@When* annotation to guide the Interpreter in choosing between the two methods. This change could be submitted as a global one, affecting current and future instances of the orchestration when the manager decides to extend their business worldwide. Similarly, the second partner could be included by changing the model as made to manage the special order from the international university, but this time applying the change globally.

```
@Action(name="scheduleDelivery")
@When("deliveryAddress.isNational()")
public void
scheduleNationalDelivery(OrderInfo order,
                         Address deliveryAddress)
{...}

@Action(name="scheduleDelivery")
@When("deliveryAddress.isInternational()")
public void
scheduleInternationalDelivery(OrderInfo order,
                              Address deliveryAddress)
{...}
```

Listing 8: New concrete actions for process evolution

4. RELATED WORK

The idea of handling exceptions and dealing with deviations is not new to the service age era. Indeed, the problems we have described in this paper share plenty of commonalities to what the research areas on *software processes* [4], and *workflow management systems* [9, 14, 18] have discovered. Both processes and service compositions are long-running, complex and dynamic entities that need to evolve and react to changes during the course of execution. Having recognized this feature, a lot of research has been conducted in order to find ways to model flexible processes and how to manage deviations originated during process enactment. This past work can be classified in three main directions:

Process programming with exceptions. A number of approaches investigated how to adapt the exception handling constructs that are supported by standard programming languages for inclusion in languages intended for process definition and automation. This direction could be illustrated by languages like APPL/A [28] and Little-JIL [20], several Workflow Management Systems [27], and by most of business process languages like BPEL and BPMN. The main drawback of this approach is that it requires all possible exceptional conditions to be identified before writing

the process code. This can be quite restrictive in highly dynamic contexts in which new and unanticipated cases may arise. This limitation is exacerbated by the fact that languages which follow this approach usually adopts a normative paradigm of modeling and a rigid runtime system, which do not allow to deviate from the model at process execution time, if something unexpected happen.

Reflective Mechanisms. Some software process execution environments—such as SPADE [5], OASIS [16], Endeavors [7], EPOS [15], and IPSE 2.5 [8]—adopted reflective languages, through which process models and even their running instances may be accessed as data items to be inspected and modified at process enactment time. As an example, the SPADE environment provides a fully reflective process modeling language (SLANG) based on Petri nets, which allows meta-programming. In SLANG one can develop a process whose objective is to modify an existing process or even an existing process instance. The potential advantage of such an approach over the previous one is clear: the process model does not need to anticipate all possible exceptional situations, since it can include the (formal) description of how the process model itself can be modified at execution-time to cope with unexpected situations. The main drawback of this approach is that it may bring further rigidity into the approach: not only the process must be modeled (or "programmed") in all detail, but so also must the meta-process, i.e., the process of modifying the model itself. For this reason, reflection is considered an effective approach to manage major exceptional situations, which require a radical departure from the originally modeled process, and particularly those situations that are expected to occur again.

Flexible Approaches. Both previous cases are based on the assumption that a precise and enforceable process model is available and there is no way to violate the prescribed process. In other terms, there is no way to treat a deviation from the process within the formal system. Reflective languages support changes to the process, but all possible changes must follow a predefined change process, i.e., again there is no way to "escape" from a fully defined, prescriptive model. The key idea to overcome this limitation was to abandon the ambitious but unrealistic goal of modeling every aspect of the process in advance, following a prescriptive style, to focus on certain constraints that should be preserved by the process, without explicitly forcing a pre-defined course of actions. This brings a great flexibility in process enactment, avoiding micro-management of every specific issue while focusing on the important properties that should be maintained. Usually, these approaches are coupled with advanced runtime systems that support the users in finding their way through the actual situations toward the process goals, while remaining within the boundaries determined by the process model. Examples of approaches that fall under this category are PROSYT [10] and ConDec [24].

This last category is also the one we mainly took as inspiration to develop DSOL, in which we abandon the imperative style followed used by most of the service composition languages to adopt a strongly declarative and flexible approach. Such flexibility was leveraged by the runtime system, as described in this paper, to simplify the definition of exception-safe service orchestrations and to allow deviations and changes during process execution.

The complexity of implementing exception-safe service orchestrations has also been recognized by part of the research community, which is proposing *Automated Service Composition* (ASC) as an alternative approach. A subset of the approaches classified under the ASC umbrella promises fully automatic construction of the orchestration from a large (potentially universal) set of semantically-rich service descriptions, to be interpreted, selected, combined, and executed by the orchestration engine, based on pre-defined initial states and goals [26]. This should happen without the intervention of a service architect, whose role is fully subsidized by the engine itself. Examples of such approaches are [22], [6] and [25]. With the workflow being generated at runtime, and using services dynamically selected, the probability of exception is minimized. In the case they happen, a re-plan mechanism, as in our approach could also be used. We believe that such approach is too ambitious, as it requires all services to be semantically described with enough details to allow the engine to choose and combine them in the right way to satisfy the users goals. Furthermore, compensation mechanism are usually ignored. We prefer to follow the mainstream path that suggests human intervention to model the service orchestration through an ad-hoc language.

Some approaches, like [13], [23] and [17], which refer to BPEL as the de-facto language for service orchestrations, have tried to extend it in order to support dynamic adaptation. Even some BPEL engines, like Apache ODE [1], have some features to allow process instances management. The problem of these approaches is that architects would still have to use BPEL as the specification language, whose adoption we have deprecated previously in Section 2.

A similar (and more general) problem has also been tackled in the research area of *Dynamic Change Management* [19, 21], i.e., systems that are able to evolve without stopping or affecting parts that are not influenced by changes. Such systems must guarantee the consistency between the old version and the new version to be deployed and also the correct completion of running instances. Our approach provides an ad-hoc solution to the problem in the specific context of DEng, as the Interpreter is able to replan as soon as a model update is detected. The re-planning phase guarantees that the new plan found is coherent with the plan that was being executed and to the current state of the orchestration. The compensation mechanisms help guarantee consistency between the current state and the state the new plan should start.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced the approach provided by DSOL and its execution engine DEng to deal with changes (forecasted or not) during service orchestration execution. First, we have shown that the concepts behind DSOL's modular declarative approach simplify the specification of alternative, or exceptional paths, anticipated at design time. The same foundation was also leveraged to simplified the implementation of changes during runtime, allowing changes to be also applied immediately to running instances, guaranteeing the consistency of on-going executions and model versions.

As future work, we plan to extend our current DEng implementation by enabling those changes to be performed through an API. This can be useful for the implementation of monitoring mechanisms, in which other programs, not only the architect, will be able to modify the orchestration model, possibly removing or disabling faulty actions before they actually fail.

Acknowledgment

This work was partially supported by the European Commission under FP7 Programme IDEAS-ERC, Project 227977–SMScom and under the "Service Architectures, Infrastructures and Engineering", Project 215483–S-Cube.

6. REFERENCES

[1] Apache ODE – Orchestration Director Engine. `http://ode.apache.org/bpel-management-api-specification.html`.

[2] Java API for XML-Based Web Services (JAX-WS) 2.0. `http://jcp.org/en/jsr/detail?id=224`.

[3] Web Services Business Process Execution Language Version 2.0, 2006. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.

[4] R. Balzer. Tolerating inconsistency. In *Proceedings of the 13th international conference on Software engineering*, ICSE '91, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[5] S. C. Bandinelli, A. Fuggetta, and C. Ghezzi. Software process model evolution in the spade environment. *IEEE Trans. Softw. Eng.*, 19:1128–1144, 1993.

[6] P. Bertoli, M. Pistore, and P. Traverso. Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence*, 174(3-4):316 – 361, 2010.

[7] G. Bolcer and R. Taylor. Endeavors: a process system integration infrastructure. In *Software Process, 1996. Proceedings., Fourth International Conference on the*, pages 76 –89, Dec. 1996.

[8] R. Bruynooghe, J. Parker, and J. Rowles. Pss: A system for process enactment. In *Proceedings of the 1st International Conference on the Software Process*, pages 128–141, Oct. 1991.

[9] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. *Data and Knowledge Engineering*, 24(3):211 – 238, 1998. ER '96.

[10] G. Cugola. Tolerating deviations in process support systems via flexible enactment of process models. *IEEE Trans. Software Eng.*, 24(11):982–1001, 1998.

[11] G. Cugola, E. Di Nitto, A. Fuggetta, and C. Ghezzi. A framework for formalizing inconsistencies and deviations in human-centered systems. *ACM Trans. Softw. Eng. Methodol.*, 5:191–230, 1996.

[12] G. Cugola, C. Ghezzi, and L. S. Pinto. Process programming in the service age: Old problems and new challenges. In *Engineering of Software*, pages 163–177. Springer Berlin Heidelberg, 2011.

[13] R. Fang, Z. L. Zou, C. Stratan, L. Fong, D. Marston, L. Lam, and D. Frank. Dynamic Support for BPEL Process Instance Adaptation. In *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 1*, pages 327–334, 2008.

[14] P. Heinl, S. Horn, S. Jablonski, J. Neeb, K. Stein, and M. Teschke. A comprehensive approach to flexibility in workflow management systems. *SIGSOFT Softw. Eng. Notes*, 24:79–88, March 1999.

[15] L. Jaccheri, J. Larsen, and R. Conradi. Software process modeling and evolution in epos. In *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering*, pages 574 –581, June 1992.

[16] P. Jamart and A. van Lamsweerde. A reflective approach to process model customization, enactment and evolution. In *'Applying the Software Process' , Proceedings of the 3rd International Conference on the Software Process*, pages 21 –32, Oct. 1994.

[17] X. Jia, S. Ying, Z. Liang, D. Xie, and J. Wen. A reflective approach for dynamic change of bpel process. *Wuhan University Journal of Natural Sciences*, 13, 2008.

[18] P. J. Kammer, G. A. Bolcer, R. N. Taylor, A. S. Hitomi, and M. Bergman. Techniques for supporting dynamic and adaptive workflow. *Computer Supported Cooperative Work (CSCW)*, 9:269–292, 2000.

[19] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16:1293–1306, November 1990.

[20] B. S. Lemer, E. K. McCall, A. Wise, A. G. Cass, L. J. Osterweil, and J. Stanley M. Sutton. Using Little-JIL to Coordinate Agents in Software Engineering. 2000.

[21] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu. Version-consistent Dynamic Reconfiguration of Component-based Distributed Systems. In *Proceedings of the 19th Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11. ACM, 2011.

[22] S. A. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 482–496, 2002.

[23] A. Mosincat and W. Binder. Transparent runtime adaptability for bpel processes. In *Proceedings of the 6th International Conference on Service-Oriented Computing*, ICSOC '08, pages 241–255, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] M. Pesic and W. van der Aalst. A Declarative Approach for Flexible Business Processes Management. In J. Eder and S. Dustdar, editors, *Proceedings of the BPM 2006 Workshops (BPD, BPI, ENEI, GPWW, DPM, semantics4ws)*, volume 4103 of *Lecture Notes in Computer Science*, pages 169–180. Springer, Springer, 2006.

[25] S. R. Ponnekanti and A. Fox. SWORD: A developer toolkit for web service composition. In *Proceedings of the 11th International WWW Conference (WWW2002)*, Honolulu, HI, USA, 2002.

[26] J. Rao and X. Su. A Survey of Automated Web Service Composition Methods. In *LNCS*, volume 3387/2005, pages 43–54. Springer, 2005.

[27] N. Russell, W. van der Aalst, and A. ter Hofstede. Workflow Exception Patterns. *Advanced Information Systems Engineering*, pages 288–302, 2006.

[28] S. M. Sutton, Jr., D. Heimbigner, and L. J. Osterweil. Language constructs for managing change in process-centered environments. *SIGSOFT Softw. Eng. Notes*, 15:206–217, October 1990.

[29] S. A. White. Business Process Modeling Notation, V1.1. Technical report, OMG, 2008.