Computing manuscript No.
(will be inserted by the editor)

DSOL: A Declarative Approach To Self-Adaptive Service Orchestrations

Gianpaolo Cugola · Carlo Ghezzi · Leandro Sales Pinto

Received: date / Accepted: date

Abstract Service Oriented Computing (SOC) has brought a simplification in the way distributed applications can be built. Mainstream approaches, however, failed to support dynamic, self-managed compositions that would empower even non-technical users to build their own orchestrations. Indeed, because of the changeable world in which they are embedded, service compositions must be able to adapt to changes that may happen at run-time. Unfortunately, mainstream SOC languages, like BPEL and BPMN, make it quite hard to develop such kind of self-adapting orchestrations. We claim that this is mostly due to the *imperative* programming paradigm they are based on. To overcome this limitation we propose a radically different, strongly *declarative* approach to model service orchestration, which is easier to use and results in more flexible and self-adapting orchestrations. An ad-hoc engine, leveraging well-known planning techniques, interprets such models to support dynamic service orchestration at run-time.

1 Introduction

When first proposed, the principles behind *Service Oriented Architectures (SOAs)* and the *Service Oriented Computing (SOC)* paradigm held the promise to solve the complexity behind programming large-scale, distributed applications. By *orchestrating* [1] existing services through easy to use languages, even non-technical users were promised to be empowered with the ability of creating their own added-value services. Unfortunately, after some years of research, technological developments, and experience, we are still far from reaching these goals: "service orchestration" is still a difficult and error-prone art that requires sophisticated skills.

Politecnico di Milano
Dipartimento di Elettronica e Informazione - DEI
Piazza Leonardo Da Vinci, 32 - 20133, Milano, Italy E-mail: {cugola,ghezzi,pinto}@elet.polimi.it

The main source of complexity has to do with the fact that service orchestrations live in a very unstable world, in which changes occur continuously and unpredictably. If not adequately managed, these changes inevitably lead to failures. As an example, orchestrations may fail because the target services they invoke have been discontinued by their providers, because they became unreachable through the currently used interconnection network, or because new versions have been deployed, which are incompatible with the previous ones. It is therefore fundamental that the orchestration could respond to such kinds of changes, finding an alternative strategy to achieve its goal.

The traditional way to achieve such kind of adaptation is by explicitly programming the orchestration and by heavily using exception handling techniques to manage failures when they occur. This is quite hard per-se and cannot be done by inexperienced users. In addition, using mainstream SOC languages, the alternative ways to achieve the orchestration's goal cannot be separate from the exception handling code. This brings further complexity and results in orchestration models that are hard to read and maintain.

We strongly argue that the main reason behind this complexity has to do with the very nature of the languages used to define service orchestrations by current mainstream SOC. Indeed, the mainstream languages used to define service orchestrations, like BPEL [2] and BPMN [3], too closely resemble traditional programming languages, with their imperative style of programming that requires service architects to take care of every aspect in the control flow among services: from the most general to the most specific ones. Service compositions must explicitly define all the different routes to go from the initial to the final state of the orchestration, and they have to forecast and explicitly manage in advance all possible faults and exceptions that may happen at run-time.

This severe limitation of the state-of-practice in service orchestration has been recognized by part of the research community, which is proposing *Automated Service Composition* (ASC) as an alternative approach. Unfortunately, we find that this alternative also has limitations. Indeed, the number of different approaches that fall under the ASC umbrella can be roughly classified in two main groups [4,5]. The first includes approaches that aim at reaching a 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. This should happen without the intervention of the service architect, whose role is fully subsidized by the engine itself. The second group of proposals is less ambitious, leaving to service architects the goal of defining an abstract model of the orchestration, which the engine interprets and makes concrete by selecting and invoking the actual services to accomplish each task. We claim that none of the two approaches completely solve our problem. The former works in specific, restricted domains, but can hardly be applied in more general settings, since 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. The latter is too restrictive, as it relies on the service architect to provide

an abstract yet detailed enough model of the orchestration, often using languages like BPEL and BPMN, whose structure is the ultimate source of the problems[1].

In this paper we illustrate a different approach. We follow the mainstream path that suggests human intervention to model service orchestrations through an ad-hoc language, but we abandon the imperative style of currently available languages in favor of a strongly declarative alternative, called *DSOL - Declarative Service Orchestration Language*. DSOL allows an orchestration to be modeled by describing: (i) a set of *abstract actions*, which provide a high-level description of the elementary activities that are typical of a given domain, (ii) a set of *concrete actions*, which map the abstract actions to the actual steps to be performed to obtain the expected behavior (typically, invoking an external service), and (iii) the *goal* of the orchestration that has to be built.

DSOL models are executed by *DEng - the DSOL Engine*, an ad-hoc service orchestration engine, which leverages automatic planning techniques to elaborate, at run-time, the best sequence of activities (i.e., service invocations) to achieve the goal. Whenever a change happens in the external environment, which prevents execution to be completed, DEng behaves in a self-healing manner. Through dynamic re-planning and advanced re-binding mechanisms it finds an alternative path toward the goal and continues executing it. DEng is fully implemented in Java and it is available for downloading[2].

In the rest of the paper we describe our approach in details and show, through a running example, the advantages it provides against the traditional SOC paradigm to support self-adaptation.

The next section introduces the example we will use throughout the paper to motivate our new approach, which we briefly describe in Section 3. Section 4 provides a detailed description of DSOL, the orchestration language we propose, and its execution environment DEng. Section 5 focuses on case studies. Finally, Section 6 discusses related work and Section 7 provides some conclusions and draws future lines of research.

2 Motivations

To illustrate the limitations of existing SOC approaches and to motivate the need for a paradigm shift like the one we propose, this section introduces an example, which we will also use throughout the paper to describe our proposal. The idea is to design a service to buy tickets for a (night) event and to arrange the transportation from the city where the user lives to the city where the event is scheduled, plus the related accommodation. Such a service is built as an orchestration of existing external services, through which one can buy the ticket, book the transportation, and book the accommodation. In particular, we consider the following requirements:

[1] We will come back to this issues in Section 6, while discussing related work.

[2] DEng is available at http://www.dsol-lang.net

1. The system shall initially ask the user to provide her relevant data, the city where she lives, the event she wants to attend, the credit card data to pay for the ticket, and the desired transportation and accommodation types.
2. Purchase of the ticket shall precede other actions, since the purpose of the trip is to participate in the event.
3. Transport purchase shall precede accommodation reservation. Transportation can be arranged either by plane, train, or bus. If the participant does not express a preference for a specific transportation type, plane, train, and finally bus will be tried, in this order.
4. Choice of accommodation shall have the following options: hotel and hostel, in this order of preference, unless explicitly chosen by the user.
5. The transportation and the accommodation must be booked for the same period. The preferred option is to book the transportation in such a way that the participant arrives at the event's location the day before the event and departs the day after, booking two nights at a nearby hotel/hostel, taking her time to visit the place. The other choice is to book the outbound transportation for the same day of the event and returning the day after, thus requiring accommodation for a single night.

Looking at the requirements of the service to implement it is clear that the overall goal can be accomplished in several ways, although there is some preferred (partial) ordering among the different actions that build the orchestration. In particular, some paths are alternative w.r.t. others (e.g., if the participant has no preference for transportation, booking a train is only required if there are no flights available), while others have to be done in sequence (e.g., the action of booking the transportation has to come before the action of booking the accommodation). Moreover, several things can go wrong at run-time: services may fail, the transportation could be available for the day before the event, but the accommodation could be available only for the night of the event, thus requiring transportation booking to be undone, and so on.

Implementing this kind of orchestration using BPEL—the de-facto standard language for service orchestration—is hard because the language adopts the imperative programming style we deprecated in the previous section, forcing service architects to explicitly code all possible action flows, and to forecast all possible exceptions. To illustrate this fact, Listing 1 shows a code snippet that expresses the alternatives for booking the transportation. The control flow relies on the fact that the external web services invoked to perform each single step return `true` if they are successfully executed, and `false` otherwise. The return value is then bound to a variable, e.g., `flightBooked`. Afterwards, the value of this variable is tested, and if it is `false` the next alternative is executed; otherwise, the other alternatives are skipped. Together with this logic we also had to code the compensation handlers to undo the effects of booking the transportation in case the accommodation for the same period could not be booked.

Although this is just a small fragment of the orchestration we consider, which is by itself quite a simple example, it is easy to see how convoluted and error prone the process of defining all possible alternative paths turns out to be. Things become even more complex when run-time exceptions, like an error in invoking an external service, enter the picture and we have to add the code to effectively manage them, e.g., by invoking alternative services. We argue that the main reasons behind these problems is that the orchestration language too closely resembles traditional imperative programming languages with their need to explicitly program the flow of execution, while the code for fault and compensation handling is mixed with the main code, further reducing the overall readability of the resulting code.

Another important point to highlight is that this approach makes it impossible to introduce new, alternative paths of actions, when during orchestration's execution something unexpected happens, such that none of the initially provided options are able to accomplish the orchestration's goal. Every possible problem has to be anticipated and managed at design time.

To mitigate these problems, we present a novel approach for service orchestration in which process activities and goals are described using a declarative language, while planning techniques are used at run-time to determine how the different tasks have to be executed to achieve the goals and how to replan in case of faults (either expected or unexpected ones). The next section overviews this approach.

3 Our Approach in a Nutshell

Service orchestrations are modeled in BPEL and BPMN as monolithic programs, which capture the entire flow of execution from the start of the orchestration to the invocation of the elementary services in charge of executing each step. DSOL adopts a radically different approach. Indeed, the DSOL service orchestration model includes different aspects, which are defined separately, possibly by different stakeholders, each bringing their own competences. Specifically, as shown in Figure 1, a service orchestration includes the following elements:

- the definition of the *orchestration interface*, i.e., the signature of the service that represents the entry point to the orchestration;
- the *goal* of the orchestration, declaratively expressed by a domain expert, not necessarily competent in software development, as a set of facts that are required to be true at the end of the orchestration;
- the *initial state*, which models the set of facts one can assume to be true at orchestration invocation time. This is described by the same domain expert who formulates the goal;
- a set of *abstract actions*, which model the primitive operations that can be invoked to achieve a certain goal and are typical of a certain domain. They are described using a simple, logic-like language that can be mastered even by non-technical domain experts;

```
...
<scope name='EventPlanning'>
 <scope name='BookTransportation'>
  <if>
   <condition>
    <!-- preferredTrans equals airplane or
         preferredTrans is null -->
   </condition>
   <scope name='BookFlight'>
    <compensationHandler>
     <!-- Cancel flight reservation -->
    </compensationHandler>
    <invoke operation='bookFlight'
            inputVariable='transportationDetails'
            outputVariable='flightBooked' .../>
   </scope>
  </if>
  <if>
   <condition>
    <!-- preferredTrans equals train or
        (preferredTrans is null and not flightBooked) -->
   </condition>
   <scope name='BookTrain'>
    <compensationHandler>
     <!-- Cancel train reservation -->
    </compensationHandler>
    <invoke operation='bookTrain'
            inputVariable='transportationDetails'
            outputVariable='trainBooked' ... />
   </scope>
  </if>
  <if>
   <condition>
    <!-- preferredTrans equals bus or
         (preferredTrans is null and not flightBooked and
          not trainBooked) -->
   </condition>
   <scope name='BookBus'>
    <compensationHandler>
     <!-- Cancel bus reservation -->
    </compensationHandler>
    <invoke operation='bookBus'
            inputVariable='transportationDetails'
            outputVariable='busBooked' ... />
   </scope>
  </if>
  <if>
   <condition>
    <!-- not (trainBooked or flightBooked or busBooked) -->
   </condition>
   <throw faultName='TransportationNotBooked' />
  </if>
 </scope>
</scope>
...
```

Listing 1 Booking transportation in BPEL

– a set of *concrete actions*, one or more for each abstract action, written by a software engineer to map abstract actions into the concrete steps required to implement the operation modeled by the abstract action, e.g., by invoking an external service or executing some code.

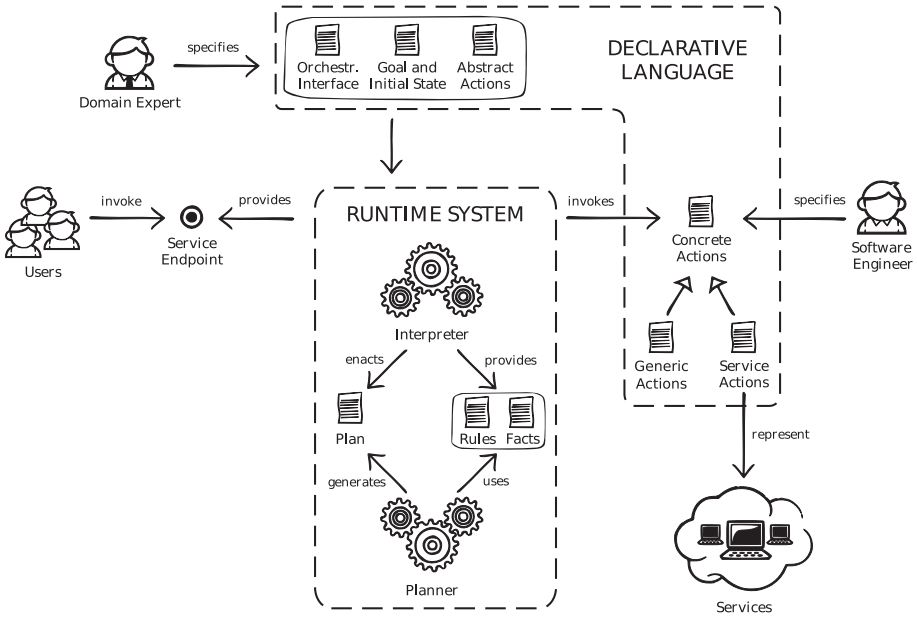

Fig. 1 The DSOL approach to service orchestration

At orchestration invocation time the *DSOL 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. 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. If something goes wrong (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, otherwise it invokes the Planner again to try a different course of action. In the extreme case, the service architect may intervene to add new abstract/concrete actions to be used to solve very complex situations.

This brief description shows the main advantages of our approach w.r.t. traditional ones:

1. We achieve a clear separation among the different aspects of an orchestration: from the more abstract ones, captured by goals, initial state, and abstract actions, to those closer to the implementation domain, captured by concrete actions.
2. We meet one of the original goals of SOC; i.e., we involve users who are not expert in software development into the cycle.
3. By focusing on the primitive actions available and letting the actual flow of execution to be automatically built at run-time through the Planner, we allow orchestration designers to focus on the general aspects that are typical of a certain domain and remain stable over time, ignoring the pecu-

liarities of a specific orchestration, which may change when requirements change. This last aspect also holds the promise to increase reusability, since the same abstract and concrete actions can be reused for different orchestrations within the same domain.
4. By separating abstract and concrete actions, with several concrete actions possibly mapped to a single abstract action, we allow the DSOL Interpreter to find the best implementation for each orchestration step and to try different routes if something goes wrong at run-time, in a fully automated way.
5. Because abstract actions only capture the general rules governing the ordering among primitive actions, the Interpreter, through a careful re-planning mechanism, can automatically overcome potentially disruptive and unexpected situations happening at run-time.
6. The modularity and dynamism inherent in the DSOL approach allow the orchestration model to be easily changed at run-time, by adding new abstract/concrete actions when those available do not allow to reach the orchestration's goal.

4 DSOL and the DEng Tool in Detail

Hereafter we provide a detailed description of the DSOL language and its orchestration engine, focusing on the dynamic adaptation capabilities they offer.

4.1 Abstract Actions

Abstract actions are high-level descriptions of the primitive actions available in a given domain, which we use 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*.

Listing 2 illustrates the abstract actions involved in modeling our Event Planning reference scenario. To clarify the structure of actions we take the `bookFlight` case as an example. The action *signature* includes its *name* and a list of *arguments*. In the example, `bookFlight(From, To, Arrival, Departure)`. The *precondition* is expressed as a list of *facts* that must be *true* in the current state for the action to be enabled. In our example we use the expressions `city(From)`, `city(To)`, `date(Arrival)`, and `date(Departure)` to constrain the values of the parameters of the action, while we use the facts `preferredTrans(airplane)` and `ticketBought` to express the fact that the `bookFlight` action must be invoked only if the user has chosen the airplane as its preferred transportation and that the ticket must have been already bought. Following the preconditions are the *postconditions*, which model the effects of the action on the current state of execution by listing the facts to be added to the state and those to be removed. In our example, when `bookFlight` is executed the fact `transportationBooked(From, To, Arrival, Departure)` is

```
action buyTicket(Event,PD)
pre: event(Event), paymentDetails(PD)
post: ticketBought

action bookFlight (From,To,Arrival,Departure)
pre: city(From), city(To), date(Arrival), date(Departure),
ticketBought, preferredTrans(airplane)
post: transportationBooked (From,To,Arrival,Departure)

action bookTrain (From,To,Arrival,Departure)
pre: city(From), city(To), date(Arrival), date(Departure),
ticketBought, preferredTrans(train)
post: transportationBooked (From,To,Arrival,Departure)

action bookBus (From,To,Arrival,Departure)
pre: city(From), city(To), date(Arrival), date(Departure),
ticketBought, preferredTrans(bus)
post: transportationBooked (From,To,Arrival,Departure)

action bookHotel (City,CheckIn,CheckOut)
pre: city(City), date(CheckIn), date(CheckOut),
at(City,CheckIn,CheckOut), preferredAccomm(hotel)
post: accommodationBooked(City,CheckIn,CheckOut)

action bookHostel (City,CheckIn,CheckOut)
pre: city(City), date(CheckIn), date(CheckOut),
at(City,CheckIn,CheckOut), preferredAccomm(hostel)
post: accommodationBooked(City,CheckIn,CheckOut)
```

Listing 2 The abstract actions for the Event Planning example

added to the state while no facts are removed (removed facts, when present, are designed using the "~" symbol).

Facts in our language are expressed as *propositions*, characterized by a name and a set of parameters. The latter represent the relevant objects of the domain. More specifically, parameters that start with an uppercase letter are considered as unbound objects and must be replaced by actual instances, i.e., those which start with a lowercase letter, to generate an execution plan. For instance, if at any point of plan generation the fact `city(event.city)` is added to the state, the object `event.city` becomes available to be bound either to the `From` or `To` generic parameters in the `bookFlight` action.

In some cases it is necessary to relate different states of a domain, e.g., to specify that when a certain situation arises new facts could be deduced. To model these situations we introduce *seam actions*. Unlike standard abstract actions, seam actions do not have a concrete counterpart, as they do not model an actual step of the orchestration, but rather a logical relation among facts in the domain. Listing 3 shows the seam actions used in the example scenario. Consider the `onTransportationBooked` case as an example. This action models the fact that after transportation is booked we may assume that the user will be at destination from the date of arrival till the date of departure. Similarly, the seam action `setTransportationPreference` models the fact that if the user does not express any preference about the transportation mode then all the three means (airplane, train, and bus) are equally possible.

```
seam action extractEventInformation
pre: event(event)
post: city(event.city), date(event.date), date(event.dayAfter),
      date(event.dayBefore)

seam action setTransportationPreference
pre: preferredTrans(empty)
post: preferredTrans(airplane), preferredTrans(train), preferredTrans(bus)

seam action setAccommodationPreference
pre: accommodation(empty)
post: accommodation(hotel),accommodation(hostel)

seam action onTransportationBooked(From,To,Arrival,Departure)
pre: transportationBooked(From,To,Arrival,Departure)
post: at(To,Arrival,Departure)
```

Listing 3 The seam actions for the Event Planning example

```
start true

goal
  ticketBought,
  transportationBooked(participantCity,event.city, event.dayBefore,
                       event.dayAfter),
  accommodationBooked(event.city, event.dayBefore,
                      event.dayAfter)
or
  ticketBought,
  transportationBooked(participantCity,event.city,
                       event.date,event.dayAfter),
  accommodationBooked(event.city,event.date,event.dayAfter)
```

Listing 4 Initial state and goal for the Event Planning example

Another important use of seam actions is to increase reusability of abstract actions coming from different models when they use different terms to express the same concept, i.e., to move from one ontology to another.

4.2 Orchestration Goal and Initial State

Besides abstract actions, the *initial state* and *goal* are also needed to produce an orchestration plan. The former models the state from which the orchestration starts, while the latter represents the desired state to reach after executing the orchestration. The goal may actually include a *set of states*, which reflect all the alternatives to accomplish the goal of the orchestration, listed in order of preference. The Planner will try to build a plan that satisfies the first goal; if it does not succeed, it will try to satisfy the second goal, and so on.

The initial state and goal of the Event Planning scenario described in Section 2 are illustrated in Listing 4. In this case the initial state is empty, while the goal models the acceptable results of our example, in the preferred order as specified in Section 2.

```
@WebService
public interface EventPlanning {
 public void plan(
  @WebParam(name="city")
  String participantCity,
  @WebParam(name="event")
  Event event,
  @WebParam(name="paymentDetails")
  PaymentDetails pd,
  @WebParam(name="preferredTrans") @Concrete
  TransportationMode transportationMode,
  @WebParam(name="preferredAccomm") @Concrete
  AccommodationType accommodationType
}
```

Listing 5 The Event Planning orchestration interface

```
buyTicket(event,pd)
bookFlight(participantCity, event.city, event.dayBefore,event.dayAfter)
bookHotel(event.city,event.dayBefore,event.dayAfter)
```

Listing 6 A possible plan for the Event Planning example

4.3 Orchestration Interface

To formalize how the orchestration is exposed as a web service, DSOL uses a Java interface properly annotated with JAX-WS [6] verbs to let the Interpreter automatically build the WSDL of the service.

The same annotations, in particular `@WebParam`, are also used by the Interpreter to create a set of additional facts to be passed to the Planner. As an example, from the orchestration interface of our reference scenario, shown in Listing 5, the Interpreter builds the facts `city(participantCity)`, `event(event)`, and `paymentDetails(pd)`, which introduce new objects to be used in generating the plan. The other two arguments of the interface are also transformed into facts for the Planner, but in a different manner. As they are annotated as `@Concrete`, their actual value (as passed by the client and transformed into a string using the `toString` Java method) is used, not the formal parameter name. Hence, if the client invokes the service with a value `airplane` for the `transportationMode` parameter, the fact `preferredTrans(airplane)` is added to the set of facts passed to the Planner. Similarly, if a `null` value is used, the fact `preferredTrans(empty)` is used. The same will happen for the `accommodationType` argument.

Using these facts, plus the goals and initial state presented in the previous section, together with the abstract and seam actions, the Planner is able to build a plan, like the one presented in Listing 6 for our reference example[3]. It includes a list of abstract actions that can lead from the initial state to a state that satisfies an orchestration goal (the first one in our case). Notice that:

[3] We omit the seam actions from the plan as they do not represent steps to be actually performed (e.g., invoking external services) at run-time.

```
@Action(name="bookFlight",service="flight")
@ReturnValue("transportationDetails")
public abstract TransportationDetails bookFlight(
        String from, String to, Date arrival, Date departure);
```

Listing 7 The `bookFlight` service action

(i) when several sequences of actions could satisfy the preferred orchestration goal, the Planner chooses one, non deterministically; (ii) although 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.

4.4 Concrete Actions

Concrete actions are the executable counterpart of abstract actions. They are intended to be specified by a different actor, i.e., a technical person with programming skills, once the abstract actions have been identified and specified by the domain expert.

In the reference implementation of the DSOL engine, concrete actions are implemented through Java methods using the ad-hoc annotation `@Action` to refer to the abstract actions they implement. 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 DSOL Interpreter haves other options to accomplish the orchestration step specified by the failed abstract action.

Among concrete actions, we distinguish between *service actions* and *generic actions*. The former are abstract methods directly mapped to external services. As an example see Listing 7. The special attribute `service` of the `@Action` annotation specifies the external service to invoke, while a hot-pluggable module of the Interpreter: (the *Service Selector* in Figure 1), 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, while the Interpreter can be easily extended to support other SOA technologies.

Unlike service actions, generic actions are ordinary Java methods to be used as utilities every time an abstract action cannot be implemented by simply invoking an external service. As an example, when the parameters of the abstract action to implement have to be pre-processed before being passed to an external service or when more than one service have to be invoked to realize the abstract action. Listing 8 shows two generic actions (code omitted for simplicity) to implement the `buyTicket` abstract action. Notice the use of the `@When` annotation to guide the Interpreter in choosing among different concrete actions for the same abstract action, based on the actual state of the orchestration.

```
@Action("buyTicket")
@When("pd.method.equals(CREDIT_CARD)")
public void buyTicketCreditCard(Event evt,
                                PaymentDetails pd){
  // Buy ticket and pay with credit card
}

@Action("buyTicket")
@When("pd.method.equals(BANK_TRANSFER)")
public void buyTicketBankTransfer(Event evt,
                                  PaymentDetails pd){
  // Buy ticket and pay with bank transfer
}
```

Listing 8 The `buyTicket` generic action

The actual state of the orchestration is represented by the abstract objects manipulated by the Planner and by the concrete (i.e., Java) objects manipulated by the Interpreter at run-time. Both are kept by the Interpreter into the *Instance Session*. This is a key-value database, which maps each abstract object used by the Planner and referenced inside the plan with a corresponding concrete object. When the orchestration is invoked, the values passed by the client are associated with the corresponding abstract objects and they are used to start populating the Instance Session. When the Interpreter must invoke a concrete action to execute the next step of the plan, it uses the Instance Session to retrieve the Java objects to pass to the action, while the value returned by the action, if any, is kept into the Instance Session, mapped to the abstract object whose name is given through the `@ReturnValue` annotation (see Listing 7 for an example). This way the abstract plan produced by the Planner is concretely executed by the Interpreter, step by step.

4.5 Failures and Compensation Actions

The ability to tolerate—both expected and unexpected—exceptions to the standard flow of actions is fundamental for a system that has to operate in an open, dynamic world. To pursue this goal, we provide both specific language constructs and ad-hoc run-time facilities.

Among the former we already mentioned the ability of associating different concrete actions to the same abstract action. This gives the Interpreter the ability to try different options to realize each step of a plan. Indeed, when an abstract action A has to be executed, the Interpreter tries the first concrete action implementing A. If this fails (e.g., it returns an exception) it tries the second one, and so on. As an example, imagine that our reference orchestration model includes two concrete actions mapped to the same abstract action `bookFlight`. The first, shown in Listing 7, invokes the external service `flight`, while the second invokes a different external service. Now suppose that the provider of the `flight` service has changed its interface. When the Interpreter executes the `bookFlight` abstract action, the concrete action it is

```
@Action(name="bookFlight",service="flight",
        compensation=true)
public abstract void cancelFlightReservation(
 @ObjectName("transportationDetails")
 TransportationDetails flightDetails
);
```

Listing 9 The `cancelFlightReservation` compensation action

bound to fails, but this does not stop the orchestration execution. In fact, the Interpreter automatically captures the exception and tries the second concrete action, which invokes a different external service, which hopefully is available and executes correctly.

If, however, none of the available concrete actions is able to execute correctly, a second mechanism is available, which involves the ability of building an *alternative plan* when something bad happens at run-time. That is, if the Interpreter is unable to realize a step (i.e., an abstract action invoked with specific parameters) of the current plan, it invokes the Planner again forcing it to avoid the failed step. This way a new plan is computed that does not include the step that was impossible to realize. 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. As an example, consider the case where the outbound flight has been booked for the day before the event (and the inbound flight for the day after) according to the plan in Listing 6, but neither a hotel nor a hostel are available the day before the event. In such a situation, the Interpreter invokes the Planner again, which produces a new plan that reaches the second goal in Listing 4. This requires the `bookFlight` action to be compensated because the new plan requires the flight to be booked for the same day of the event.

Since the design of *compensating actions* usually requires application level knowledge, DSOL allows service architects to explicitly define them for each defined concrete action. Listing 9 shows how to compensate the `bookFlight` action. We notice how compensation actions use the same syntax of concrete actions, with the special `compensation=true` attribute. By default, compensation actions are invoked using the same parameters of the action to compensate. For those cases where it is necessary to use different parameters, they can be taken from the Instance Session using the `@ObjectName` annotation, as shown in Listing 9, which invokes the compensation action `cancelFlightReservation` with the `TransportatioDetails` returned by the `bookFlight` action it undoes.

Notice how in this example we used a further mechanism provided by DSOL to increase robustness of the orchestration, namely the ability to specify multiple goals for each orchestration (see Section 4.2). This opportunity has been leveraged by the Planner at run-time, to build a new plan that automatically bypasses the failed step.

In summary, by combining the ability to specify different implementations (i.e., concrete actions) for each step of a plan, with the ability to rebuild failed plans in search of alternative courses of actions, possibly achieving different, still acceptable goals, our language and run-time system allow robust orchestrations to be built in a natural and easy way. Indeed, by combining these mechanisms, DSOL orchestrations are able to automatically get around failures and any other form of unexpected situation, by self-adapting to changes in the external environment.

This goal is also achieved thanks to the DSOL approach to modeling orchestrations, which focuses on the primitive actions typical of a given domain more than on the specific flow of a single orchestration. This approach maximizes the chance that when something bad happens, even if not explicitly anticipated at modeling time, the actions that may overcome the situation have been modeled and are available to the Planner and Interpreter.

4.6 The DEng Tool

DEng, the DSOL execution engine, is organized into three main components: the Interpreter, the Planner, and the Service Selector.

The Interpreter is the core of the tool, being responsible for the execution of the orchestration. Essentially, it replies to requests coming from the clients by interpreting the orchestration modeled in DSOL, with the help of the Planner and Service Selector.

The Interpreter is built on top of Apache CXF [7], an open-source service framework. Apache CXF supports all the service-related parts inside DEng. It is responsible for building the service model (i.e., the WSDL of the orchestration) from the orchestration interface specified in DSOL. Furthermore, it handles all the communication between external clients and the orchestration, including support for several transport protocols, e.g. HTTP, Servlet, and JMS, and a variety of web service specifications including WS-Addressing, WS-Policy, WS-ReliableMessaging, and WS-Security. Apache CXF is also used to invoke the SOAP-based services that implement DSOL's service actions, as it provides a dynamic way to invoke such services, without the need for creating stub and data classes in advance. Conversely, to invoke RESTful services the Interpreter leverages the Apache Http Components [8], which support the different HTTP methods.

When a client submits a request to a DSOL orchestration, the Interpreter forwards the request to a class that implements the orchestration interface. This generic implementation is automatically built by the Interpreter at runtime, using the Code Generation Library (CGLIB) [9]. Its role is to read the meta-data of the called method and the actual values of the parameters and, based on this information, build the initial state of the orchestration and initiate the Instance Session with the key-value pairs that correspond to the actual parameters. Then, it retrieves the abstract actions and goals associated with the invoked method and passes them to the Planner.

The Planner is a hot-pluggable component of DEng, which is accessed by the Interpreter through a generic interface. The current DEng prototype uses an ad-hoc planner, built as an extension of JavaGP [10,11], an open-source implementation of the Graphplan [12] planner. The JavaGP planner was extended to support multiple goals and the possibility of setting the initial state of the plan at run-time. The JavaGP planner was also modified to introduce the ability of inhibiting the use of some steps in the plan, i.e., those that in the DSOL model are mapped to concrete actions whose invocation failed.

Using the initial state and the abstract actions provided by the Interpreter, the Planner builds a plan to reach the specified goal and returns it to the Interpreter, which enacts it by linking each step of the plan to the concrete actions that implement it. Classes that contain concrete actions (methods annotated with `@Action`) are parsed once for all at orchestration start-up, and they are stored in memory using a Hash Map binding the name of the each abstract action with the list of concrete actions (i.e., Java methods) that implement it. In such a way, when a step of the plan needs to be executed, all available implementations can be retrieved in constant time.

To actually invoke concrete actions, the Interpreter parses the next step of the plan extracting the name of the abstract action involved and the names of the objects to pass as parameters. The former is used to retrieve the concrete actions to invoke from the Map mentioned above, while the latter are used to retrieve, from the Instance Session, the actual object instances to be used as parameters. Notice that concrete actions are invoked one by one until one is found that completes successfully. The return value of such concrete action is stored into the Instance Session, making it available to the following step of the orchestration.

As explained in Section 4.5, if none of the concrete actions completes successfully, the Interpreter starts a new interaction with the Planner in order to find a new plan. In this interaction, the Interpreter first informs the Planner of the steps that cannot be used any more (those corresponding to abstract actions for which no concrete actions are available) and then it requests a new plan. When the new plan is built, the Interpreter compares it with the old one to figure out if they have steps in common that have already been executed and if there are actions that have to be compensated. The corresponding compensation actions are executed before proceeding with the new plan.

Notice that some of the concrete actions to be invoked when enacting a plan may be "service actions" (see Section 4.4) modeled as properly annotated abstract Java methods. For such methods an implementation is created at runtime using the previously mentioned CGLIB. In particular, the Interpreter uses the meta-data of the called method to request the Service Selector for services that match the given identifier. Once the services are selected, they are invoked using the dynamic clients front-end provided by Apache CXF in case of SOAP-based services, and using Http Components in case of RESTful services. The parameters used to invoke the service are the same passed to the concrete action (which were retrieved from the Instance Session as explained

above). The object returned by the service, if any, is used as a return value of the concrete action and it is stored into the Instance Session for later use.

As it happens for the Planner, the Service Selector was also designed to be a hot-pluggable component that can be replaced according to specific needs of the users. Currently, the Service Selector is implemented as a database of service descriptions, which can be managed at run-time through a specific API that allows new services to be added or existing ones to be removed. For each SOAP-based service known to the engine, this database holds information such as the service endpoint (WSDL), the port and the operation that must be invoked. For RESTful services, it holds the service endpoint (URL), the HTTP method to be used (GET, POST, PUT, DELETE) and the media type (our current prototype supports both XML and JSON) used to exchange messages.

Finally, service orchestrations written in DSOL can be deployed in two different ways. First they can be packaged as a Web application to be deployed in any Servlet container, such as Tomcat or Jetty. The second option is to use an embedded server, running as a standalone application. The first option has to be preferred when the orchestration is part of a Web application that includes its own HTML GUI as a front-end. The second option is easier to use as it relies on our DEng prototype only, not requiring any external component.

5 Case Studies and Evaluation

This section is divided in three parts. The first takes various examples of orchestrations from the literature and compares their implementation in DSOL and BPEL. The second does the same but taking AO4BPEL [13] and JOpera [14] as a reference. The goal is to show the advantages of DSOL w.r.t. two mature research proposals that address the same limitations of BPEL that we address with DSOL. The last part of the section focuses on performance, showing how the use of a planner to automatically decide the actual flow of the orchestration at run-time in practice is not a threat to scalability.

5.1 Comparison with BPEL

To reinforce the benefits provided by DSOL we take several examples from the literature and study how they can be implemented in DSOL and the differences between DSOL and a state-of-the-practice language such as BPEL.

5.1.1 The DoodleMap Example

DoodleMap [15] is a poll service used to make choices about places, with a map view of the selected locations. DoodleMap is built by composing the services provided by Yahoo! Local Search [16], Doodle poll service [17], and Google Static Maps [18]. The Yahoo! Local Search service is used to find places in a given location based on some criteria; for example, find available restaurants

in the center of Milan. The results are used as choices for the Doodle poll service, which will create the poll, and as markers for the Google Static Maps, which will create and return an image map including the selected places.

At first, one could say that DoodleMap is a simple example but we argue that this is true only because it was not designed to be fault tolerant. For example, the service provided by Yahoo! Local Search only works if the queried location is inside the United States, taking the whole service to fail when the user searches for a location abroad. To overcome this limitation, one could replace this service by the one provided by Google Places [19] that works for more countries. However, there are situations in which even Google Places could fail, for example if it is temporarily unavailable or if it returns an empty list of places. In general, one way to implement a fault tolerant orchestration is to provide alternatives for the involved services, possibly in a straightforward manner. Accordingly, in our example we will include both Yahoo! Local Search and Google Places.

Unfortunately, the APIs provided by these two services are not compatible, so this situation cannot be addressed as a late binding problem. Yahoo! Local Search can be invoked directly with the search criteria and the location as provided by the user, while Google Places does not accept a textual location but needs geographic coordinates as the reference for the search. This is a common situation: including an alternative to a service also implies changing the workflow to introduce new accessory services, which were not originally required but are now needed to correctly execute the replacing service. In our case, before invoking Google Places we have to invoke a geocoding service to translate the location provided in human-readable form by the user in geographic coordinates, e.g., Yahoo! PlaceFinder [20] or Google Geocoding [21].

Listing 10 illustrates how the scenario presented above would be written in BPEL, applying the traditional approach that achieves fault tolerance through exception handling mechanisms. A Boolean variable is declared (line 3) to determine whether Google Places (together with Google Geocoding) should be used or not. Initially the value of this variable is set to `false` and it is changed to `true` as soon as the fault handler (lines 6-15) attached to the scope `Yahoo` (lines 5-17) is enabled. This will happen when the Yahoo! Local Search service fails (line 16). Notice how the code, despite the simplicity of the example, is hard to read and how it would increase in complexity as we start adding other alternatives to this and the other services that are part of the overall orchestration. Furthermore, the order in which the alternatives are invoked, i.e., the fact that Yahoo! Local Search has to be preferred to Google Places, is hard-coded into the orchestration and it is relatively difficult to change.

Listing 11 illustrates how the same sub-scenario would be written in DSOL, with a major focus on the abstract actions involved. The action `findAvailablePlacesByLocation` models services in which places are searched by location, as Yahoo! Local Search, while the action `findAvailablePlacesByCoordinates` models the case in which nearby places are located based on geographic coordinates, e.g. Google Places. Finally, the action `getCoordinates` models services that take

```
<scope>
 <variables>
  <variable name='invokeGooglePlaces' type='xsd:boolean'/>
 </variables>
 <scope name='Yahoo'>
  <faultHandlers>
   <catchAll>
    <assign>
     <copy>
      <from>true()</from>
      <to variable='invokeGooglePlaces' />
     </copy>
    </assign>
   </catchAll>
  </faultHandlers>
  <!-- invoke Yahoo! Local Search -->
 </scope>
 <if name='GooglePlaces'>
  <condition>$invokeGooglePlaces</condition>
  <sequence>
   <!-- invoke Google Geocoding -->
   <!-- invoke Google Places -->
  </sequence>
 </if>
<scope>
```

Listing 10 Alternatives implemented in BPEL for the DoodleMap scenario

a human-readable location and transform it in geographic coordinates. If the goal of this part of the orchestration is to find the list of available places, which we could model with the DSOL fact `listofplaces(availablePlaces)`, the Planner may satisfy it in two ways: searching places directly by location or first transforming the location into coordinates and then searching places using these coordinates. DEng will try the former route first and, if it fails, it would try the second one. All this happens at run-time and it is fully automatic: the domain expert focused on the available alternatives without the need for explicitly programming the exception handling code.

As for the order in which the alternatives are tested, it can be left unspecified or it can be explicitly modeled, through an accurate use of the goal. As an example, one could use the facts `bylocation` and `bycoordinates` provided by the abstract actions `findAvailablePlacesByLocation` and `findAvailablePlacesByCoordinates`, respectively, to write a goal (see Listing 12) that lists, as two different subgoals, the two alternatives, thus making the preferred order among them explicit: first try to find places by location, then by coordinates.

Another important point to highlight is how easily the DSOL code can be reused. Imagine a variant of the original DoodleMap example in which the user is equipped with a GPS device and the location to use is the current one. To address this case we have to change the orchestration interface and the overall workflow. The former receives the location as geographic coordinates instead of using a string, while the latter can now invoke the Google Places directly but it needs a reverse-geocoding service before invoking Yahoo! Local Search. In DSOL, this variant of the original orchestration can be modeled by fully reusing the actions part of the original model and adding the new abstract

```
action findAvailablePlacesByLocation(Location, Query)
pre: searchLocation(Location), searchQuery(Query)
post: listofplaces(availablePlaces), bylocation

action getCoordinate(Location)
pre: searchLocation(Location)
post: searchCoordinate(Coordinate)

action findAvailablePlacesByCoordinate(Coordinate,Query)
pre: searchCoordinate(Coordinate), searchQuery(Query)
post: listofplaces(availablePlaces), bycoordinate
```

Listing 11 Alternatives implemented in DSOL for the DoodleMap scenario

```
goal
  listofplaces(availablePlaces), bylocation
or
  listofplaces(availablePlaces), bycoordinate
```

Listing 12 Using multiple goals to define ordering of actions

```
action getLocation(Coordinate)
pre: searchCoordinate(Coordinate)
post: searchLocation(location)
```

Listing 13 Reverse geocoding abstract action

action shown in Listing 13. It is the Planner that chooses which actions to invoke and in which order to satisfy the goal of the original orchestration or the goal of the new one.

Continuing with the DoodleMap example, we must now use the returned list of places to invoke the poll and the map service. As they are not related to each other, they can be invoked in parallel. However, before invoking those services, we need to transform the list of places into a list of choices compatible with the poll service and a list of markers compatible with the map service. Besides this, as the poll service is a stateful service it needs to be undone (i.e., compensated) if for any reason the whole orchestration fails.

In BPEL, as shown by Listing 14, all these issues have to be addressed together. The `<flow>` statement (lines 7-23) is used to specify that activities related to the map and poll services must be invoked in parallel, while the `<sequence>` statement (lines 9-12 and 18-21) indicates that the inner activities must be executed in a sequence. Lines 15-17 define the compensation handler for the `Doodle` scope, which is activated by the fault handler (lines 2-6) attached to the outer scope, which, in turn, catches all the faults that might occur as the orchestration is executed. The result is a rather convoluted piece of code that would become really unmanageable if we had included the different service alternatives available.

The DSOL approach instead separates the various aspects. The fact that actions can be executed in parallel or have to be executed in sequence is

```
1<scope>
2 <faultHandlers>
3  <catchAll>
4   <compensateScope target="Doodle" />
5  </catchAll>
6 </faultHandlers>
7 <flow>
8  <scope name="Map">
9   <sequence>
10   <!-- convert places into markers -->
11   <!-- invoke map service -->
12  </sequence>
13 </scope>
14 <scope name="Doodle">
15  <compensationHandler>
16   <!-- delete created poll -->
17  </compensationHandler>
18  <sequence>
19   <!-- convert places into poll choices -->
20   <!-- invoke poll service -->
21  </sequence>
22 </scope>
23 </flow>
24</scope>
```

Listing 14 Flow control mixed with compensation activities

deduced by the Interpreter based on the pre- and postconditions of the various actions part of the plan. While executing the plan, the Interpreter invokes actions as soon as all of their preconditions are true, i.e., all the actions of a plan whose precondition is true are invoked in parallel. Instead if an action A_1 requires a fact f to be true before starting and f is asserted as a postcondition of an action A_2 then A_2 and A_1 will be executed in this order. The service architect does not need to worry about ordering and parallelism, which are deduced by the Interpreter looking at the model itself.

Similarly, alternative ways to execute the same action can be coded by listing different concrete actions for the same abstract action, or different abstract actions with the same pre- and postconditions. Finally, compensation actions are written separately, as a special type of a concrete action. Ad-hoc annotations (see below) are used to identify them as compensation actions and to specify which action they compensate. It is the Interpreter's responsibility to decide, at run-time, if and when compensation actions have to be invoked.

If we look at our DoodleMap example, Listing 15 illustrates the abstract actions that need to be included to cover the new part of the orchestration. We notice that pre- and postconditions of `createMarkers` and `createMapWithMarkers` are put in relation by the predicate `listOfMarkers(...)`, denoting the need of running the two actions in sequence, if they appear in the same plan, while the precondition of `createMarkers` and `createOptions` are the same, making these two actions eligible for parallel execution, if they are part of the same plan.

Similarly, Listing 16 illustrates some of the concrete actions that implement the aforementioned abstract actions. Lines 3-8 represent a *generic action* that

```
action createMarkers(Places)
pre: listofplaces(Places)
post: listofmarkers(markers)

action createMapWithMarkers(Markers)
pre: listofmarkers(Markers)
post: map(mapWithMarkedPlaces)

action createOptions(Places)
pre: listofplaces(Places)
post: listofoptions(options)

action createPoll(InitiatorName, PollTitle, Options)
pre: initiatorName(InitiatorName), title(PollTitle),
     listofoptions(Options)
post: poll(poll)
```

Listing 15 New abstract actions for the DoodleMap orchestration

```
1 @Action
2 @ReturnValue('options')
3 public List<Option> createOptions(List<Place> places) {
4  List<Option> options = new List<Option>();
5  for (Place place : places) {
6   Option option = new Option(place.getName());
7   options.add(option);
8  }
9  return options;
10 }
11
12 @Action(service = 'poll')
13 @ReturnValue('pollId')
14 public abstract String createPoll(String initiatorName,
15                                    String pollTitle,
16                                    List<Option> options);
```

Listing 16 Some concrete actions for DoodleMap scenario

```
@Action(name = 'createPoll',
        service = 'deletePoll', compensation = true)
public abstract void deletePoll(@ObjectName('pollId')
                                String poll);
```

Listing 17 Compensation action for the `createPoll` action

implements the abstract action `createOptions` and transforms the available places into options to the poll. Lines 14-16 represent a *service action* that implements the abstract action `createPoll` and is executed by invoking the service labeled with the key `poll` as defined in the `@Action` annotation. The returned object could be referenced in the future through the key `pollId`.

Listing 17 also illustrates the compensation action for `createPoll`. It can be identified as a compensation action because the `compensation` attribute in the `@Action` annotation is set to `true`. It receives the poll's id returned when the poll was created as a parameter and is executed by invoking the `deletePoll` service.

```
...
<if name='MapProvider'>
 <condition>
  $doodleMapRequest.mapProvider = 'google'
 </condition>
 <!-- use Google Maps -->
 <else>
  <if>
   <condition>
    $doodleMapRequest.mapProvider = 'bing'
   </condition>
   <!-- use Bing Maps -->
  </if>
 </else>
</if>
...
```

Listing 18 Alternatives based on a service parameter using BPEL

```
action createMapUsingGoogle(Places)
pre: listofplaces(Places), mapProvider(google)
post: map(mapWithMarkedPlaces)

action createMapUsingBing(Places)
pre: listofplaces(Places), mapProvider(bing)
post: map(mapWithMarkedPlaces)
```

Listing 19 New abstract actions for the DoodleMap orchestration

The last case we consider is that of an alternative that depends on a user's choice. In DoodleMap we can imagine that the map could be created using Google Maps or Bing Maps [22] based on the preferences of the user. The traditional approach to solve this case, and the one used in BPEL, is based on nested `if` statements or `switches`. This can be another source of complexity and another factor that limits reusability of the orchestration code, as its control flow is hardwired in the code. DSOL addresses the same case by using the actual value of the parameters passed to the orchestration as facts that become part of the Initial State (see Section 4.3) and can be used into the precondition of the various actions to decide which one has to be used. Listings 18 and 19 show the BPEL and DSOL code for this case, respectively.

As a final remark, Table 1 shows a comparison between DSOL and BPEL in terms of *lines of code* (LOC). The interpretation of LOC as a quality index is rather controversial and a meaningful statistical sample should be examined in order to support any conclusion. However, on the one side, we can say that the value of LOC for different languages to implement the same functionality is an indication of the level of abstraction of the language. Table 1 indicates that DSOL provides a higher abstraction level. On the other side, LOC can be seen as an indicator of the effort needed to develop a piece of software (see [23]).The example suggests that DSOL might indeed help simplify development and reduce the required effort.

estimate effort (in terms of lines of code – LOC) required to implement the entire DoodleMap example using BPEL and DSOL. From this perspective we can conclude that DSOL not only provides an easier to use and more flexible approach to model service orchestrations, but it also simplifies the development, minimizing the effort for service architects.

LANGUAGE	CODE	#LOC
DSOL	Concrete Actions	42
	Orchestration Interface	12
	Initial State, Goals and Abstract Actions	22
	TOTAL	**76**
BPEL	Process	214
	Orchestration Interface (WSDL)	47
	TOTAL	**261**

Table 1 A comparison (in term of LOC) of DSOL and BPEL when used to model the DoodleMap orchestration

5.1.2 Other Examples

We repeated our exercise of comparing DSOL with BPEL by implementing four more scenarios: the Event Planning service, the Loan Approval service, the ATM service, and the Trip Reservation Service. The first is the example we presented in Section 2, which we implemented both in DSOL and BPEL. The last three were taken from the JBoss jBPM-BPEL v1.1.1 documentation [24]. In these examples, we took the BPEL implementations from the JBoss distribution and re-implemented them in DSOL.

We will not examine all of them in details as we did for the DoodleMap example, since the general considerations would be very similar. Rather, here we focus on the size of the resulting model. Tables 2 to 5 report the results we obtained. In all scenarios, the size of the BPEL code is between 2.2 and 4 times bigger than the code required by DSOL. The saving in size is bigger when the examples become more complex, with various alternatives in the execution flow.

5.2 Alternative Approaches

In the last years, several alternatives to BPEL were proposed by researchers to address some of the issues we emphasized in this paper. In this section we describe two of them: namely AO4BPEL and JOpera, which we chose as mature representatives of the state of the art in the area. Other systems will be reviewed in Section 6.

LANGUAGE	CODE	#LOC
DSOL	Concrete Actions	54
	Orchestration Interface	16
	Initial State, Goals and Abstract Actions	38
	TOTAL	**108**
BPEL	Process	381
	Orchestration Interface (WSDL)	53
	TOTAL	**433**

Table 2 A comparison (in term of LOC) of DSOL and BPEL when used to model the Event Planning orchestration

LANGUAGE	CODE	#LOC
DSOL	Concrete Actions	36
	Orchestration Interface	19
	Initial State, Goals and Abstract Actions	12
	TOTAL	**75**
BPEL	Process	140
	Orchestration Interface (WSDL)	44
	TOTAL	**184**

Table 3 A comparison (in term of LOC) of DSOL and BPEL when used to model the Loan Approval orchestration

LANGUAGE	CODE	#LOC
DSOL	Concrete Actions	81
	Orchestration Interface	46
	Initial State, Goals and Abstract Actions	40
	TOTAL	**167**
BPEL	Process	369
	Orchestration Interface (WSDL)	116
	TOTAL	**485**

Table 4 A comparison (in term of LOC) of DSOL and BPEL when used to model the ATM orchestration

5.2.1 Aspect-Oriented Extensions to BPEL

To increase modularity of orcheatration models and to better support their run-time adaptation, some researchers proposed to use Aspect-Oriented Programming (AOP) techniques. AO4BPEL [13] and BPEL'n'Aspects [25] are two notable representatives of this class of systems, which are built around an aspect-oriented extension to BPEL. In this section we focus on the former as it addresses both modularity and run-time adaptability as we do with DSOL.

In particular, in [13] the authors introduce various examples to illustrate how BPEL lacks tools to properly modularize *crosscutting concerns* among several processes. Here we take them and we show how DSOL behaves in those cases.

LANGUAGE	CODE	#LOC
DSOL	Concrete Actions	56
	Orchestration Interface	22
	Initial State,	
	Goals and Abstract Actions	42
	TOTAL	**120**
BPEL	Process	201
	Orchestration Interface (WSDL)	66
	TOTAL	**267**

Table 5 A comparison (in term of LOC) of DSOL and BPEL when used to model the Trip Reservation orchestration

```
//Actions
action invokeS1(..)
pre: ..
post: S1Invoked

action countS1Invocations
pre: S1Invoked
post: counterForS1Incremented

//Goal
goal(.. and S1Invoked and counterForS1Incremented)
```

Listing 20 New abstract action and goal for the "data collection for billing" example

The first example presented in [13] (*data collection for billing*) assumes that a service provider starts charging a fee for using its Web Service S_1. The client, who will receive a bill from the provider, wants to check whether the bill is accurate. This requires counting how many times S_1 has been called from any deployed orchestration. In BPEL one would need to examine all the deployed orchestrations, finding out where the service is invoked, and manually including there the code to invoke a counting Web Service. AO4BPEL solves this problem more elegantly by declaring a single aspect to be executed after S_1, which invokes the counting Web Service.

Looking at this example from the DSOL perspective, however, we notice that what was hard to modularize in BPEL (the crosscutting concern) can be easily integrated into a single module in DSOL. Indeed, in DSOL the various orchestrations invoking the original Web Service S_1 would share a single abstract action, similar to action `invokeS1` in Listing 20. To count how many times such action has been invoked one could introduce a new action `countS1Invocations` (whose pre-condition is the post-condition of the `invokeS1`), changing the goals of the involved orchestrations to include the post-condition of the new action as shown in Listing 20[4]. This way the counting code is inserted only once and it is the Planner which guarantees that it is included into all plans that included the original action.

[4] The concrete action associated with action `countS1Invocations`, which actually invokes the counting service, is straightforward and has been omitted.

```
@ReturnValue("transportationDetails")
@Action(name="bookFlight")
public TransportationDetails PersistentBookFlight(
        String from, String to, Date arrival, Date departure){
  TransportationDetails flightDetails = bookFlight(from,to,arrival,departure);
  DAO.saveData(flightDetails);
}

@ReturnValue("transportationDetails")
@Action(name="bookTrain")
public TransportationDetails PersistentBookTrain(
        String from, String to, Date arrival, Date departure){
  TransportationDetails trainDetails = bookTrain(from,to,arrival,departure);
  DAO.saveData(trainDetails);
}

@Action(service="flight")
public abstract TransportationDetails bookFlight(
        String from, String to, Date arrival, Date departure);

@Action(service="train")
public abstract TransportationDetails bookTrain(
        String from, String to, Date arrival, Date departure);
```

Listing 21 The modified `bookFlight` and `bookTrain` concrete actions including data persistence

The second example presented in [13] (*data persistence*) moves from the consideration that in BPEL all data elaborated during an orchestration is lost as soon as the orchestration ends. In several scenarios, discarding the orchestration data is not an acceptable behavior. For instance, in the Event Planning scenario, the payment confirmation code, the booking confirmation for the hotel, and the flight details should be stored. In BPEL, the solution for such problem would be similar to the solution presented for the *data collection for billing* example. The code to keep the desired data persistent would not be modularized in one place but replicated in different parts of different orchestrations. Again, AO4BPEL addresses such a situation by modularizing the persistency code into a single aspect that intercepts the calls for a given activity and stores the desired data for later use.

As in the first example, the peculiar approach to modeling orchestrations taken by DSOL makes the data persistence aspect a well modularized one. Indeed, DSOL distinguishes between the abstract, high-level model of an orchestration (abstract actions and goals) and its implementation (the concrete actions), leaving the actual flow of execution to be decided at run-time. The data persistency policy can be considered an implementation aspect to being modeled by introducing ad-hoc concrete actions as in Listing 21, which shows the original `bookFlight` and `bookTrain` concrete actions of the Event Planning examples, and their persistent counterparts that use an external data access object (DAO) [26] to persist data after invoking the original actions.

The third use case for AOP presented in [13] is about *business rules*. The authors do not make specific example, but claim that in general business rules are hard to modularize in BPEL and are amenable to be modeled as an aspect

```
@Around("call(@org.dsol.annotation.Action * *(..))")
public Object executionTimeMonitor(ProceedingJoinPoint thisJoinPoint) {
  long start = System.currentTimeMillis();
  Object returnObject= thisJoinPoint.proceed();
  long end = System.currentTimeMillis();
  long executionTime = end - start;
  ...
  return returnObject;
}
```

Listing 22 Example on how AspectJ could be integrated with DSOL

that intercepts some activities and encodes the business rule in a single place. As in the previous examples, DSOL does not suffer from this problem. Its rule-based nature allows to encode most business rules easily as part of the various abstract/concrete actions, while the fact that the orchestration flow is derived at run-time by the Planner starting from the available actions and the goal, guarantees that the appropriate actions (i.e., the appropriate business rules) are included into every plan when they are required. As a further justification of this claim we notice that the same authors of AO4BPEL, in their paper [27] focus explicitly on the issue of appropriately modeling business rules and sketch two solutions considered equivalent: one based on AOP, the other based on a rule-engine that operates in a way similar to our Planner.

The final example presented in [13] concerns the *measurement of activity execution time and logging*. We acknowledge that this is the example that least fits DSOL and is also the one that mostly benefits from AOP, in our opinion. While we do not have a general solution for this case, we observe that DSOL and DEng are ultimately based on Java, so it is not hard to integrate them with one among the many available aspect-oriented extensions of Java, such as AspectJ [28]. For example, Listing 22 illustrates an AspectJ pointcut that intercepts all calls for a method annotated with `@Action` and calculates its execution time.

Besides modularity, AO4BPEL also claims that aspects can be used to solve the problem of dynamic changes and process evolution at runtime. To do so, aspects could be used to add, at runtime, new activities in specific points of the process. If business rules changes, the only thing to do is to activate or deactivate the appropriate aspect. We claim, however, that such an approach is not adequate for changing an orchestration. First, the changes are limited to the declared aspects, ignoring the fact that it is often necessary to change or remove also the activities that were initially declared into the orchestration. Furthermore, using aspects to evolve the orhectration means that changes, instead of being incorporated as a natural evolution of the model are realized more as a sort of patches, which could even complicate the overall understandability and maintainability of the orchestration.

In DSOL, changes are handled in a complete different way [29], since the modularity and dynamism inherent in the DSOL approach provide support for ad-hoc mechanisms to change the orchestration at runtime. Indeed, as the

plan of execution, i.e., the actual sequence of activities to be performed, is built at runtime, changing the orchestration is much simpler in DSOL compared to the complex mechanisms that other, more traditional systems, must put in place to obtain the same result. 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. When a new model for an existing orchestration is submitted, one can specify if it has to affect future executions, the 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.

5.2.2 JOPERA

JOpera [14] is a mature research product that offers a visual language and a fully functional execution plataform for building distributed application composed of reusable services, which includes, but is not limited to Web Services. The graphical and visual approach offered by JOpera simplifies modeling complex orchestrations when compared with BPEL. Moreover, the fact that the control flow and the data flow of the composition are described separately, allows one to build orchestrations that are easier to understand and maintain. On the other hand, the overall style of the language is still largerly procedural and consequently it suffers from most of the flexibility problems we highlighted in this paper.

Figures 2 and 3 illustrate, respectively, the control flow and the data flow diagrams for the DoodleMap scenario described in Section 5.1.1. These descriptions are inherently rather complex. As in BPEL, service orchestrations in JOpera must be modeled in all details, forcing the architect to decide and forecast at design time the best alternatives and the order in which such alternatives must be tried. Fault tolerance must be explicitly programmed by heavily using exception handling mechanisms. For example, the arrow with a red dot end that connects activity `YahooLocal` with activity `GoogleGeocode` means that `GoogleGeocode` must be executed after `YahooLocal` if the latter fails. Similarly, the question mark that annotates some activities denotes that they are guarded by some condition. For example, `CreateGoogleMap` and `CreateBingMap` depend on the user's choice. In DSOL all these features are handled automatically by the Planner, facilitating the job of the architect and allowing the orchestration to evolve more easily. Moreover, although the JOpera user does not need to specify some details, like, for example, which activities have to be performed in order and which ones can be done in parallel (a detail that is deduced by the execution engine from the data flow diagram), the process structure remains quite complex to define and rigid to evolve.

Although the graphical formalism provided by JOpera and the textual one provided by DSOL are not easy to compare, we contend that the fine-grain, imperative modeling imposed by JOpera leads to readability and maintain-

ability issues that do not apply to the DSOL solution, which benefits from a declarative approach that focuses on the relevant details of the orchestration, while other aspects, including the actual flow of execution, are decided at run-time. As a comparison, DSOL requires 76 lines of code and 7 abstract actions in total to encode the same example reported in Figures 2 and 3.

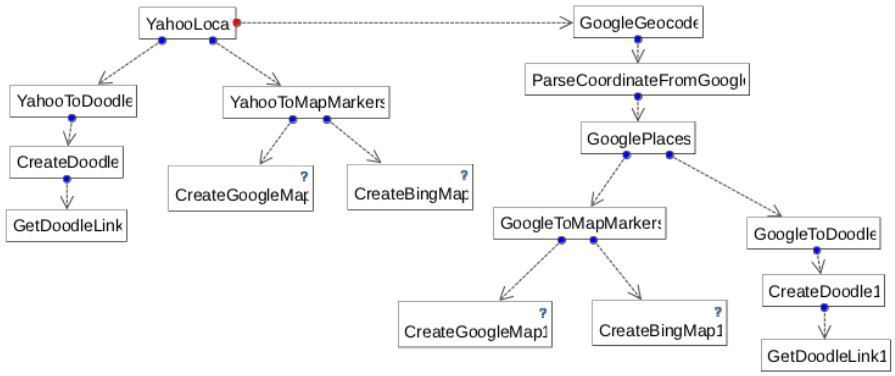

Fig. 2 DoodleMap control flow modeled in JOpera

5.3 Empirical Assessment

In the previous sections we focused on the expressiveness and usability of our modeling language DSOL. Here instead we are interested in testing if and how using a planner to decide the actual flow of the orchestration at run-time may negatively impact performance. To do so, we developed an application that automatically generates different sets of related abstract actions and goals and we used the Planner to extract a plan from these data. To test the performance of the Planner under different situations, we varied the number of abstract actions that are part of the model and their structure, i.e., the number of parameters they have and the complexity of their pre and postconditions. For preconditions, we also distinguished between the predicates that include some of the action's parameters (e.g., predicate `listofplaces(Places)` in the precondition of abstract action `createMapUsingGoogle` from Listing 19), from those that are pure (fully bound) facts that must be true for the action to be called (e.g., predicate `mapProvider(google)` in the same abstract action). The same distinction was made for postconditions, distinguishing between facts that involve some of the action's parameters (e.g., fact `listofplaces(availablePlaces)` in the postcondition of abstract action `findAvailablePlacesByLocation`), from those that are fully bound (e.g., fact `byLocation` in the same abstract action).

We considered a base scenario characterized by the following parameters: 50 abstract actions, 4 predicates in each precondition and postcondition (2

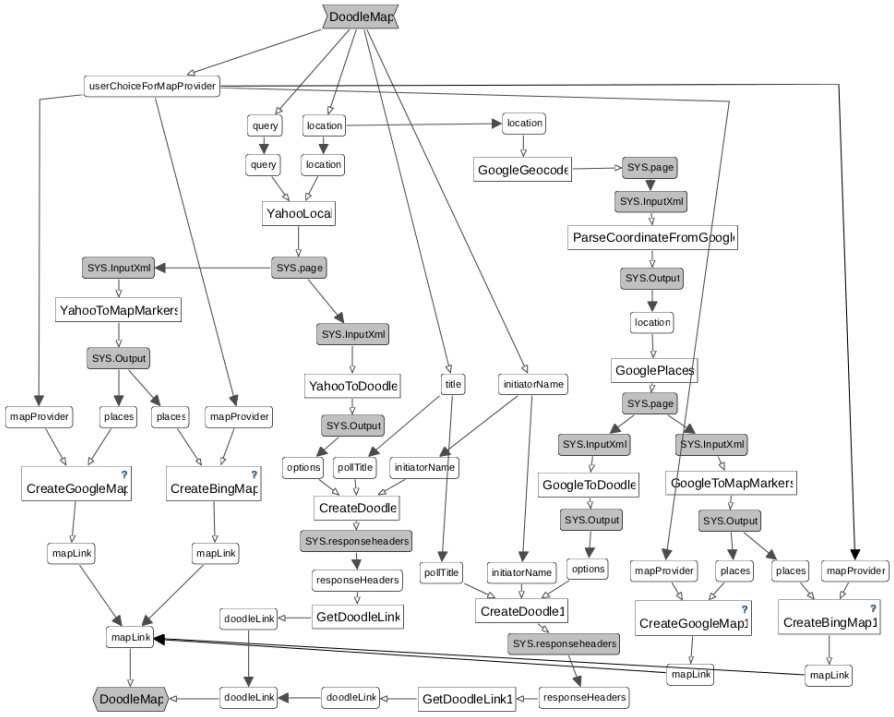

Fig. 3 DoodleMap data flow modeled in JOpera

of one type and 2 of the other). As for the goal, it was chosen in a way that plans with different sizes would be generated[5]: 5, 10, and 15. We repeated each experiment 30 times and plotted the average result we measured and the 95% confidence interval.

Our evaluation was carried out for a server deployed in the Amazon cloud, configured as a small instance [30] and running Ubuntu Linux 10.10. This environment was set up to emulate a typical configuration used to deploy service orchestrations in real scenarios.

Figure 4 shows the results we measured in the base scenario. They show the feasibility of our approach as the time to create the plan is very reasonable. For instance, if we consider the case of plans of size 10 (i.e., involving 16 abstract actions, on average) it only takes 250 ms to the Planner to build such plans. If only one third of the 16 actions are calls of external services, the overhead caused by external interactions will dominate the overhead imposed by the Planner. Even in the case of building a rather complex plan including 26 abstract actions on average, with size of 15, requires only 550 ms of the Planner.

[5] The size of the plan differs from the actual number of abstract actions that compose it, as some of the actions can be executed in parallel. For plans of size 5, 10, and 15 the mean number of abstract actions in the plan is 8, 16, and 26, respectively.

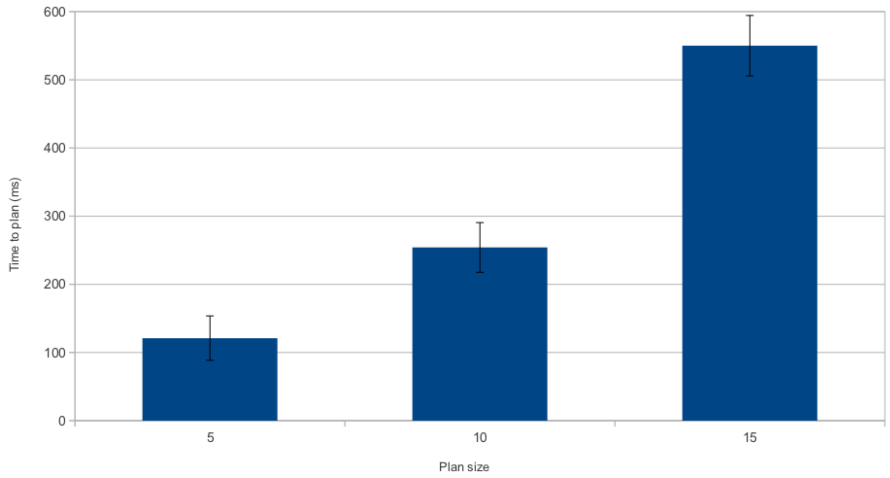

Fig. 4 Time required to build plans of different sizes for the base case

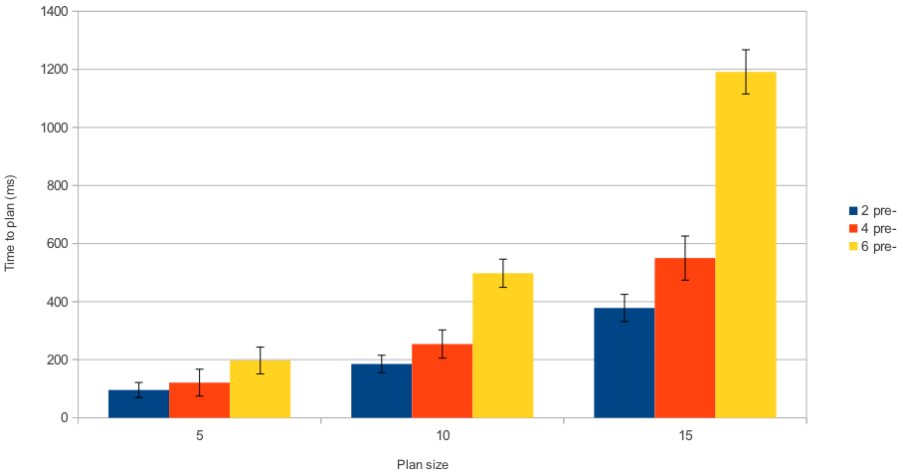

Fig. 5 Time required to build plans of different sizes varying the number of predicates in preconditions

Figures 5 and 6 show how the complexity of preconditions impacts on the time required to build the plan. In Figure 5, we consider the base scenario but we change the number of predicates in preconditions from 2 to 6. In Figure 6, we keep the number of predicates in preconditions fixed(4) and change the ratio of the two types of predicates involved. At one extreme all the predicates reference one of the action's parameters (they are unbound predicates), while at the other extreme they are all bound. While a growing number of predicates in preconditions and particularly a growing number of the unbound ones has a negative impact on the plan building time, this remains acceptable, with a max value of 3.5 sec and an average value just above 1 sec. Again, we expect these

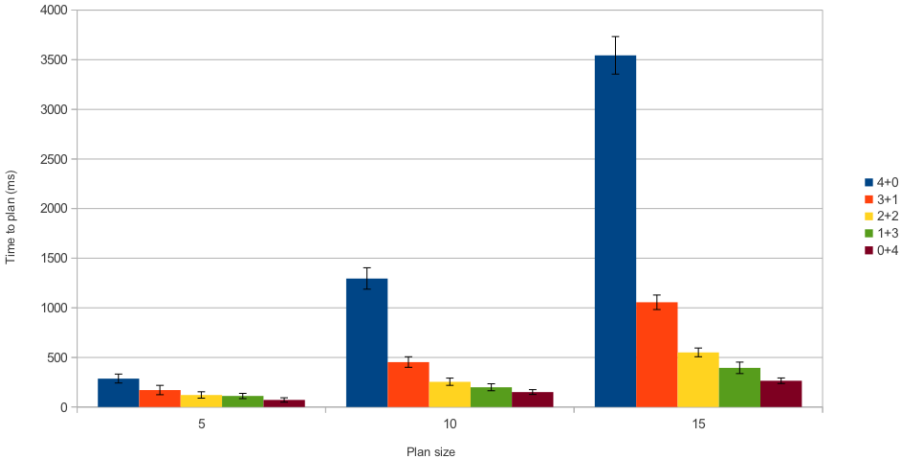

Fig. 6 Time required to build plans of different sizes varying the proportion between the types of predicates in preconditions

values to be dominated by the time required to invoke the external services that build such complex orchestrations.

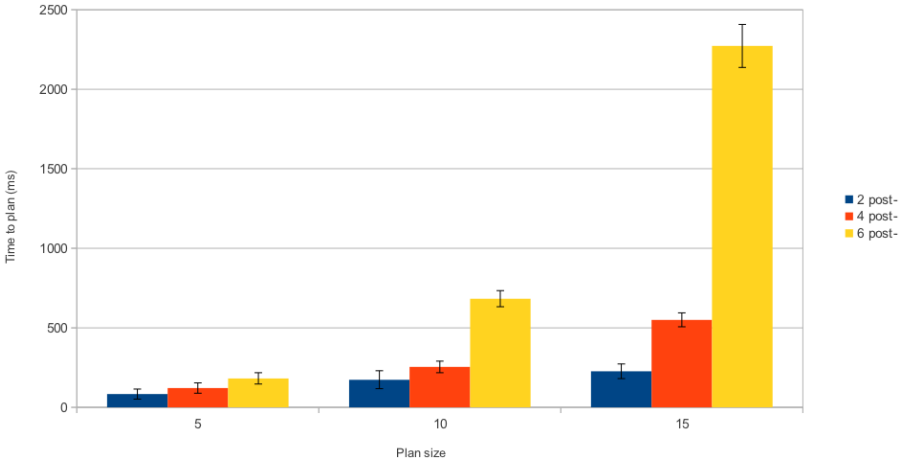

Fig. 7 Time required to build plans of different sizes varying the number of facts in post-conditions

Figures 7 and 8 make a similar analysis on postconditions. Notice that in Figure 8, the case in which none of the facts in postconditions refers to the actions' parameters is not possible because it would lead to an empty plan. The results are similar to the previous case, with a worst case of 4 sec and an average of 1.5.

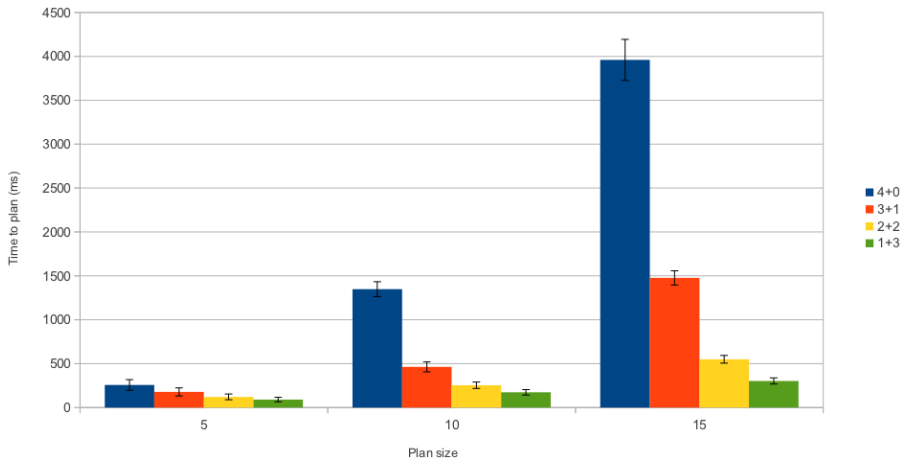

Fig. 8 Time required to build plans of different sizes varying the proportion between the two types of facts in postconditions

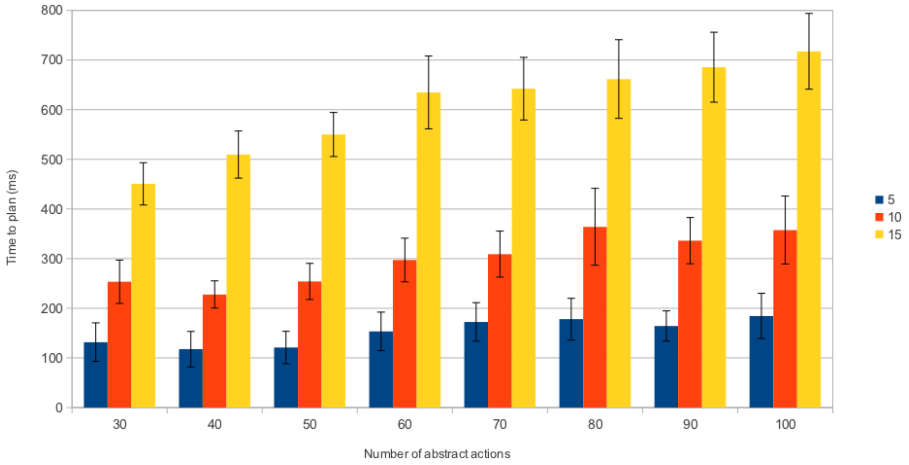

Fig. 9 Time required to build plans of different sizes varying the number of available abstract actions

Figure 9 shows the time required to build a plan in our base scenario but changing the number of available abstract actions from 30 to 100 (they were 50 in the base scenario). Our Planner scales very smoothly here, with a time that is almost constant and always below 1 sec. Starting from this positive result, we decided to investigate what happens in the worst case in which there is no plan to reach the goal. Figure 10 shows our base scenario, with a growing number of available abstract actions and with specially chosen goals that cannot be satisfied. Here we see that the number of available actions impacts on the number of combinations to test before concluding that no plan can be built. In the worst case (100 abstract actions) the Planner takes more

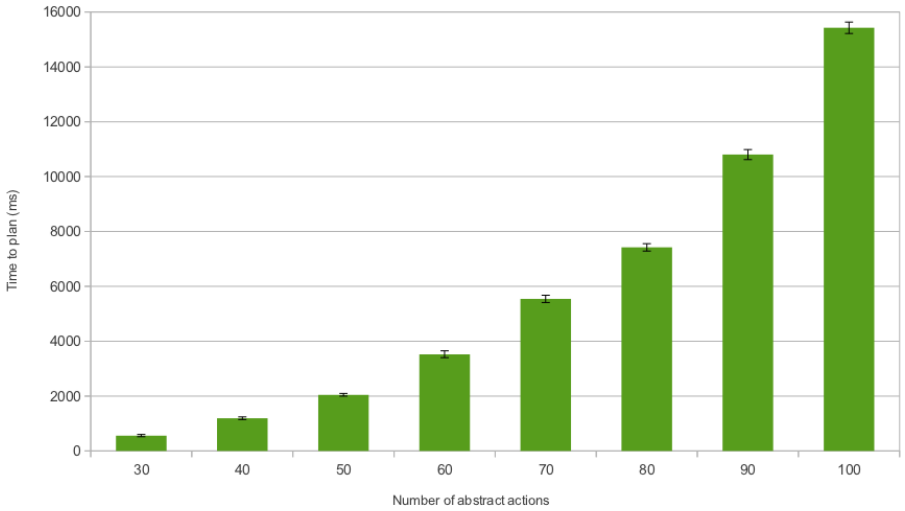

Fig. 10 Time required to discover that a goal is not satisfiable in scenarios characterized by a growing number of abstract actions

than 15 sec to decide that no plan may reach the goal. This is a negative result but it has been obtained in a fairly tough scenario: 100 actions and no plan. In a more reasonably sized scenario like our default one (50 actions), the worst case of no plan requires only 2 sec to be solved.

In general, from the above assessment we may conclude that our approach is feasible and the use of a Planner introduces an acceptable (often negligible) overhead in the execution time of the overall orchestration. To further confirm this statement, we compared the overall performance of our DEng run-time, from planning to actual execution of the orchestration, with one of the most widely adopted BPEL engines: ActiveBPEL [31]. In particular, we compared the time required to complete the whole DoodleMap orchestration (see Section 5.1.1) including the time to invoke the various web services involved, observing the system from a client's perspective. We invoked the orchestration with different inputs to test different paths of execution (including those that require DEng to initially build a plan that will fail, with the need to re-plan at run-time), and repeated our tests multiple times to account for the variations that may come from invoking the external services.

Engine	Time (ms)
DSOL	1571
ActiveBPEL	1543

Table 6 Performance comparison between DEng and ActiveBPEL

Table 6 shows the results we obtained when the orchestration is invoked to build a DoodleMap for pizza restaurants in New York. Yahoo! Local Search

completes successfully in this scenario and the orchestration is executed without faults. Both engines require approximately the same time to complete the orchestration, with DEng being slightly slower (by less than 30 ms).

Engine	Time (ms)
DSOL	2579
ActiveBPEL	2130

Table 7 Performance comparison between DEng and ActiveBPEL in presence of faults that require re-planning

Table 7 shows the results obtained when the orchestration is invoked with pizza restaurants in Milan as the key parameter. The Yahoo! Local Search fails as it can only handle requests for locations in the USA. In BPEL this failure triggers the fault handler that enables the orchestration to use the Google Places service to find the set of available locations. In DSOL, this will force the Interpreter to invoke the Planner once again to build the new plan that includes the Google Places related action. Again, the response time is similar, with DEng being slower by a greater but still acceptable margin of 450 ms.

Engine	Time (ms)
DSOL	2075
ActiveBPEL	1873

Table 8 Performance comparison between DEng and ActiveBPEL

Finally, Table 8 shows the average time considering both alternatives together. In this scenario, the impact of our planning-based approach, plus the overhead introduced by the other parts of our engine, which make many runtime decisions using late-binding at each step, from the choice of the actual services to invoke, to the choice of the concrete actions to use, to the choice of the flow of execution itself, affect performance for 200 ms, i.e., 10% of the total time. We argue that this is an acceptable price to pay for the flexibility that it brings.

6 Related Work

During the last years, various proposals have been made to reduce the complexity inherent in defining service compositions, with the goal of further increasing the diffusion of this technology. Hereafter, we review those that are mainly related with our work.

As an alternative to BPEL and BPMN in the specification of service compositions, other languages, like JOpera [14] (see our specific comparison in Section 5.2.2), Jolie [32], and Orc [33], were proposed. While easier to use and often more expressive than BPEL and BPMN, they do not depart from

the imperative paradigm, and consequently they share with them the same problems that motivate our work.

To overcome these limitations, other researchers followed the idea of adopting a declarative approach. Among those proposals, DecSerFlow [34,35] is the closest to our work. In DecSerFlow service choreographies are defined as a set of actions and the constraints that relate them. Both actions and constraints are modeled graphically, while constraints have a formal semantics given in Linear Temporal Logic (LTL). There are several differences between DecSerFlow and DSOL. First of all, DecSerFlow focuses on service choreographies and on modeling them to support verification and monitoring. Conversely, we focus on service orchestrations and specifically on enacting them. This difference motivates the adoption of LTL as the basic modeling tool, as it enables powerful verification mechanisms but introduces an overhead that can be prohibitive for an enactment tool [34]. The DSOL approach to modeling offers less opportunities for verification but it can lead to an efficient enactment tool. Secondly, DSOL emphasizes re-planning at run-time as a mechanism to support self-adaptive service orchestrations that maximize reliability even in presence of unexpected failures and changes in the external services. This is an issue largely neglected by DecSerFlow, as it focuses on specification and verification and it does not offer specific mechanisms to manage failures at run-time.

GO-BPMN [36–38] is another declarative language, designed as a Goal-Oriented extension for traditional BPMN. In GO-BPMN business processes are defined as a hierarchy of goals and sub-goals. Multiple BPMN plans are attached to the "leaf" goals. When executed, they achieve the associated goal. These plans can be alternative or they can be explicitly associated to specific conditions through guard expressions based on the context of execution. Although this approach also tries to separate the declarative statements from the way they can be accomplished, the alternative plans to achieve a goal must be explicitly designed by the service architect and are explicitly attached to their goals. The engine does not automatically decide how the plans are built or replaced; it just chooses between the given options for each specific goal, and it does so at service invocation time. The DSOL ability to build the plan dynamically and to rebuild it if something goes wrong at run-time, improves self-adaptability to unexpected situations.

The approach described in [39] defines a goal-oriented service orchestration language inspired by agent programming languages, like AgentSpeak(L) [40]. One of the main motivations of this approach is the possibility of following different plans of execution in the presence of failures. The main difference with our approach is that the alternative plans need to be explicitly programmed based on the data stored into the Knowledge Base and the programmer needs to explicitly reason about all the possible alternatives and how they are related, in a way similar to that adopted by traditional approaches. In the presence of faults, the facts that compose the Knowledge Base are programmatically updated to trigger the execution of specific steps that have to be specified in advance to cope with that situation. No automatic re-planning is supported.

As briefly introduced before, the complexity in defining Web service compositions is also being tackled through *Automated Service Composition* (ASC) approaches. While our research was motivated by the desire of overcoming the limitations of mainstream orchestration languages in terms of flexibility and adaptability to unexpected situations, ASC is grounded on the idea that the main problem behind service orchestration is given by the complexity in selecting the right services in the open and large scale Internet environment. The envisioned solution is to provide automatic mechanisms to select the right services to compose, usually based on a precise description of the semantics of the services available.

For example, in [41], user requirements and Web services are both described in DAML-S [42], a semantic Web service language, and linear logic programming is used to automatically select the correct services and generate a BPEL or DAML-S process that represents the composite service. Similarly, [43] presents an extension of Golog, a logic programming language for dynamic domains, to compose and execute services described in DAML-S, based on high-level goals defined by users. Both approaches requires the exact semantics of services to be defined formally (e.g., in DAML-S) and they do not support dynamic redefinition of the orchestration at run-time to cope with unexpected situations.

Similar considerations hold for those ASC proposals that adopt planning techniques similar to those adopted in DSOL. In these approaches the planning domain is composed by the semantically described services and goals are defined by end-users. For example, [44] uses the SHOP2 planner to build compositions of services described in DAML-S. Similarly, [45] proposes an algorithm, based on planning via model-checking, that takes an abstract BPEL process, a composition requirement and a set of Web services also described in BPEL and produces a concrete BPEL process with the actual services to be invoked. In SWORD [46], the to-be composed services are described in terms of their inputs and outputs, creating the "service model". To build a new service the developer should specify its input and output, which SWORD use to decide which services should be chosen and how to combine them. XSRL, a language to express service requests, is presented in [47]. Users can use this language to specify how services should be chosen for a given request. A planner is responsible for choosing the services based on the specified request, augmenting an abstract BPEL process with the selected services.

Other ASC approaches start from an abstract "template process", expressed either in BPEL, e.g., [48,49], or as a Statechart, e.g, [50] and, taking into consideration QoS constraints and end-user preferences, select the best services among those available to be actually invoked. As mentioned in the Introduction, these approaches focus on a relatively simpler problem than DSOL, as they focus on "selecting the right services at run-time", leaving to the service architect the (complex) task of defining the abstract "workflow" to follow. Moreover, as they use traditional, procedural languages as the tool to model this abstract workflow, they suffer from the limitations and problems that we identified in Section 2. Moreover, most of the ASC approaches

proposed so far operate before the orchestration starts, while DSOL includes advanced mechanisms to automatically adapt the orchestration to the situations encountered at execution time. This is particularly evident if we consider the problem of compensating actions to undo some already performed steps before following a different workflow that could bypass something unexpected. A problem that, to the beast of our knowledge, is not considered by any of these approaches.

As a final notice, we observe that the three-layered architectural model for self-management described by [51] and [52] was also used as an inspiration for DSOL and its engine. In particular, the layers defined by this architecture are: the *goal management layer*, responsible for the generation of plans from high-level goals (in our approach, the Planner); the *change management layer*, which is concerned with using the generated plans to construct component configurations and direct their operation to achieve the goal addressed by the plan (in our approach, the DSOL Interpreter, which interacts with the Planner and executes the generated plan); at last, the *component layer*, which includes the domain specific components (in our approach, the abstract/concrete actions, used to build and enact the plan).

7 Conclusions and Future Work

In this paper we presented an approach to overcome the limitations of currently available service orchestration environments, in particular when failures occur and the orchestration needs to self-adapt to unexpected situations. This approach is based on a new language called DSOL, which models orchestration declaratively, focusing on the set of available activities, without having to explicitly declare the control-flow of the orchestration, which is generated at run-time through an ad-hoc planner, part of the DEng DSOL engine. This simplifies the task of modeling complex orchestrations, increases the level of reusability, and in case of failures can achieve self-adaptation.

Several mechanisms are part of DSOL to build self-adapting orchestrations. First, each activity is modeled through an abstract description coupled with several concrete implementations, to be tried in case of failures. If this is not enough, the Planner can be re-invoked during process execution to find an alternative way to accomplish the orchestration goal, by-passing those activities that cannot be successfully executed, and undoing already executed activities of the old plan, if necessary. At last, as no explicit workflow is pre-defined, new activities (i.e., abstract and concrete actions) can be added to the model at run-time, without the need to redeploy the entire orchestration.

For the future, we envision different lines of work. First, we want to increase the expressiveness of our modeling language by adding quality of service (QoS) facilities into DSOL. In fact, in the current version of DSOL, if different abstract actions can be used to build different plans to reach the same goal, those actions are chosen non-deterministically. We want to change this by allowing the user to express QoS requirements, used by the Planner to choose the best

sequence of actions to reach the orchestration goal with the desired QoS. A similar QoS-based approach can also be used to choose the best concrete action to invoke for each abstract action.

Moreover, we plan to build a (graphical) tool, possibly integrated in an IDE like Eclipse, to further simplify the definition of abstract actions, goals, and orchestration interfaces.

As for the DEng run-time system, while the current prototype is fully operational (and downloadable), we think there is still space to further improve performance and also to improve reliability and robustness. Another aspect that we want to improve is the support to monitoring running orchestrations. We currently catch faults as they happen and we start our counter-measures (invoking alternative concrete actions or re-building the plan), but we are not able to "anticipate" faults. More advanced monitoring mechanisms may try to anticipate faults, e.g., by checking for the actual availability of external services in advance, before their unavailability impacts the running orchestration.

We also plan to extend DEng to improve the run-time support for adding new abstract and concrete actions to a running orchestration, allowing the service architect to easily intervene to overcome the most complex exceptions that may happen during a long running orchestration.

Finally, to fully validate our approach we want to test its feasibility in additional, real world case studies.

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.

References

1. T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.
2. A. Alves, A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, D. Konig, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, A. Yiu, eds., Web Services Business Process Execution Language Version 2.0, Tech. rep., OASIS (2006). URL `http://www.oasis-open.org/apps/org/workgroup/wsbpel/`
3. S. A. White, Business Process Modeling Notation, V1.1, Tech. rep., OMG (2008). URL `http://www.bpmn.org/Documents/BPMN_1-1_Specification.pdf`
4. R. Maigre, Survey of the tools for automating service composition, in: Web Services (ICWS), 2010 IEEE International Conference on, 2010, pp. 628 –629. doi:10.1109/ICWS.2010.72.
5. J. Rao, X. Su, A Survey of Automated Web Service Composition Methods, in: LNCS, Vol. 3387/2005, Springer, 2005, pp. 43–54. URL `http://www.springerlink.com/content/4m6w37g0jffk9bv4`
6. Java API for XML-Based Web Services (JAX-WS) 2.0, `http://jcp.org/en/jsr/detail?id=224`.
7. Apache CXF: An Open-Source Services Framework, `http://cxf.apache.org/`.
8. Apache HttpComponents, `http://hc.apache.org/`.

9. Code Generation Library, http://cglib.sourceforge.net/.
10. F. Meneguzzi, M. Luck, Leveraging new plans in AgentSpeak(PL), in: M. Baldoni, T. C. Son, M. B. van Riemsdijk, M. Winikoff (Eds.), Proceedings of the Sixth Workshop on Declarative Agent Languages, 2008, pp. 63–78.
11. JavaGP - Java GraphPlan, http://emplan.sourceforge.net.
12. Graphplan, http://www.cs.cmu.edu/~avrim/graphplan.html.
13. A. Charfi, M. Mezini, Ao4bpel: An aspect-oriented extension to bpel, World Wide Web 10 (2007) 309–344. doi:10.1007/s11280-006-0016-3.
    URL http://dl.acm.org/citation.cfm?id=1285732.1285748
14. C. Pautasso, G. Alonso, Jopera: A toolkit for efficient visual composition of web services, Int. J. Electron. Commerce 9 (2005) 107–141.
    URL http://portal.acm.org/citation.cfm?id=1278095.1278101
15. C. Pautasso, Composing restful services with jopera, in: International Conference on Software Composition, Vol. 5634, Springer, Zurich, Switzerland, 2009, pp. 142–159.
16. Yahoo! Local Search Web Services, http://developer.yahoo.com/search/local/V3/localSearch.html.
17. Doodle APIs, http://doodle.com/about/APIs.html.
18. Google Static Maps API, http://code.google.com/apis/maps/documentation/staticmaps/.
19. Place Searches, http://code.google.com/apis/maps/documentation/places/#PlaceSearches.
20. Yahoo! Place Finder, http://developer.yahoo.com/geo/placefinder/.
21. The Google Geocoding API, http://code.google.com/apis/maps/documentation/geocoding/.
22. Bing Maps APIs, http://msdn.microsoft.com/en-us/library/dd877180.aspx.
23. M. J. Carlo Ghezzi, D. Mandrioli, Fundamental of Software Engineering, 2nd Ed., Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
24. JBOSS jBPM, http://www.jboss.org/jbpm.
25. D. Karastoyanova, F. Leymann, Bpel'n'aspects: Adapting service orchestration logic, in: Web Services, 2009. ICWS 2009. IEEE International Conference on, 2009, pp. 222 –229. doi:10.1109/ICWS.2009.75.
26. Data access object pattern, http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html.
27. A. Charfi, M. Mezini, Hybrid web service composition: business processes meet business rules, in: Proc. of the 2nd International Conference on Service Oriented Computing, ICSOC '04, ACM, New York, NY, USA, 2004, pp. 30–38. doi:http://doi.acm.org/10.1145/1035167.1035173.
    URL http://doi.acm.org/10.1145/1035167.1035173
28. The AspectJ Project, http://www.eclipse.org/aspectj/.
29. L. Sales Pinto, G. Cugola, C. Ghezzi, Dealing with changes in service orchestrations, in: To appear on the Proceedings of the 2012 ACM Symposium on Applied Computing, SAC '12, 2012.
30. Amazon EC2 Instance Types, http://aws.amazon.com/ec2/instance-types/.
31. ActiveBPEL, http://www.activebpel.com.
32. F. Montesi, C. Guidi, R. Lucchi, G. Zavattaro, Jolie: a java orchestration language interpreter engine, Electron. Notes Theor. Comput. Sci. 181 (2007) 19–33. doi:10.1016/j.entcs.2007.01.051.
    URL http://portal.acm.org/citation.cfm?id=1268072.1268121
33. D. Kitchin, A. Quark, W. Cook, J. Misra, The orc programming language, in: Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems, FMOODS '09/FORTE '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 1–25.
34. M. Montali, M. Pesic, W. M. P. v. d. Aalst, F. Chesani, P. Mello, S. Storari, Declarative specification and verification of service choreographies, ACM Trans. Web 4 (2010) 3:1–3:62.
35. W. van der Aalst, M. Pesic, Decserflow: Towards a truly declarative service flow language, in: M. Bravetti, M. Nunez, G. Zavattaro (Eds.), Web Services and Formal Methods, Vol. 4184 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 1–23.

36. D. Greenwood, G. Rimassa, Autonomic goal-oriented business process management, in: Autonomic and Autonomous Systems, 2007. ICAS07. Third International Conference on, 2007, pp. 43–48. doi:10.1109/CONIELECOMP.2007.61.
37. B. Burmeister, M. Arnold, F. Copaciu, G. Rimassa, Bdi-agents for agile goal-oriented business processes, in: AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2008, pp. 37–44.
38. M. Calisti, D. Greenwood, Goal-oriented autonomic process modeling and execution for next generation networks, in: MACE '08: Proceedings of the 3rd IEEE international workshop on Modelling Autonomic Communications Environments, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 38–49.
39. M. B. Van Riemsdijk, M. Wirsing, Using goals for flexible service orchestration: a first step, in: AAMAS'07/SOCASE'07: Proceedings of the 2007 AAMAS international workshop and SOCASE 2007 conference on Service-oriented computing, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 31–48.
40. A. S. Rao, AgentSpeak(L): BDI agents speak out in a logical computable language, in: MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996, pp. 42–55.
41. J. Rao, P. Küngas, M. Matskin, Composition of semantic web services using linear logic theorem proving, Inf. Syst. 31 (2006) 340–360. doi:10.1016/j.is.2005.02.005.
URL http://portal.acm.org/citation.cfm?id=1140593.1140600
42. M. H. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. V. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, K. P. Sycara, Daml-s: Web service description for the semantic web, in: Proceedings of the First International Semantic Web Conference on The Semantic Web, Springer-Verlag, London, UK, 2002, pp. 348–363.
URL http://portal.acm.org/citation.cfm?id=646996.711291
43. S. A. McIlraith, 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), 2002, pp. 482–496.
44. Automating DAML-S Web Services Composition Using SHOP2.
45. P. Bertoli, M. Pistore, P. Traverso, Automated composition of web services via planning in asynchronous domains, Artificial Intelligence 174 (3-4) (2010) 316 – 361.
46. S. R. Ponnekanti, A. Fox, SWORD: A developer toolkit for web service composition, in: Proceedings of the 11th International WWW Conference (WWW2002), Honolulu, HI, USA, 2002.
47. A. Lazovik, M. Aiello, M. Papazoglou, Planning and monitoring the execution of web service requests, Int. J. Digit. Libr. 6 (2006) 235–246. doi:10.1007/s00799-006-0002-5.
48. D. Ardagna, B. Pernici, Adaptive service composition in flexible processes, Software Engineering, IEEE Transactions on 33 (6) (2007) 369 –384. doi:10.1109/TSE.2007.1011.
49. R. Aggarwal, K. Verma, J. Miller, W. Milnor, Constraint driven web service composition in meteor-s, in: Proceedings of the 2004 IEEE International Conference on Services Computing, IEEE Computer Society, Washington, DC, USA, 2004, pp. 23–30.
URL http://portal.acm.org/citation.cfm?id=1025130.1026125
50. L. Zeng, B. Benatallah, A. H.H. Ngu, M. Dumas, J. Kalagnanam, H. Chang, Qos-aware middleware for web services composition, IEEE Trans. Softw. Eng. 30 (2004) 311–327. doi:10.1109/TSE.2004.11.
URL http://portal.acm.org/citation.cfm?id=987527.987638
51. J. Kramer, J. Magee, Self-managed systems: an architectural challenge, in: FOSE '07: 2007 Future of Software Engineering, IEEE Computer Society, Washington, DC, USA, 2007, pp. 259–268. doi:http://dx.doi.org/10.1109/FOSE.2007.19.
52. D. Sykes, W. Heaven, J. Magee, J. Kramer, From goals to components: a combined approach to self-management, in: SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems, ACM, New York, NY, USA, 2008, pp. 1–8. doi:http://doi.acm.org/10.1145/1370018.1370020.