Model Driven Service Composition

Bart Orriëns, Jian Yang, and Mike. P. Papazoglou

Tilburg University, Infolab
PO Box 90153, 5000 LE, Tilburg, Netherlands
{b.orriens,jian,mikep}@kub.nl

Abstract. The current standards for web service composition, e.g. BPEL. neither cater for dynamic service composition nor for dynamic business configuration. Our firm belief is that business processes can be built dynamically by composing web services in a model driven fashion where the design process is controlled and governed by a series of business rules. In this paper we examine the functional requirements of service composition and introduce a phased approach to the development of service compositions that spans abstract definition, scheduling, construction and execution. Subsequently, we analyze the information requirements for developing service compositions by identifying the basic elements in a web service composition and the business rules that are used to govern the development of service compositions.

1 Introduction

The platform neutral nature of web services creates the opportunity for enterprisers to develop business processes by using and combining existing web services, possibly offered by different providers. By selecting and combining the most suitable and economical web services, business processes can be generated dynamically by observing the changing business conditions.

Current composite web service development and management solutions are very much a manual activity, which require specialized knowledge and take up much time and effort. This applies even to applications that are being developed on the basis of available standards, such as BPEL4WS [4] or BPML [2]. Due to a vast service space to search, a variety of services to compare and match, and different ways to construct composed services service composition is simply too complex and too dynamic to handle manually. To automate the development of service compositions, we require a systematic way of analyzing their requirements and modelling the activities involved in them just as we do with software development. The benefits of adapting a service development methodology for service composition is that we gain much more insight in the process of constructing service compositions so that we can better manage their implementations.

In this paper we use a model driven approach to facilitate the development and management of dynamic service compositions. The central notion in this

approach entails separation of the fundamental composition logic from particular composition specifications (e.g., BPEL and BPML) in order to raise the level of abstraction. This allows rapid development and delivery of service compositions based on proven and tested models, as such supporting the service composition life-cycle. The proposed approach uses UML as the method for modelling service compositions. This will enable us to develop technology independent composition definitions, which can subsequently be mapped to a specific standard (e.g. BPEL) automatically. Furthermore, in addition to UML we use the Object Constraint Language (OCL) [9] to express business rules that govern and steer the process of service composition.

Business rules are precise statements that describe, constrain and control the structure, operations and strategies of a business. They can express e.g. pricing and billing policies, quality of service, process flow - where they describe routing decisions, actor assignment policies, etc - regulations, and so on. In current web service technology solutions, such rules are deeply embedded in the implementation of processes, leaving the user with little empowerment to manage and control them and eventually the processes themselves. When business rules are relative to business processes, these statements should be extracted from the application code in order to be more easily managed (defined and versified) and consistently executed. Our thesis is that we can use business rules to determine how a service composition should be structured and scheduled, how the services and their providers should be selected, and how service binding should be conducted. This paves the way towards developing dynamic service compositions.

The paper is structured as follows: In Section 2 we explain the functional requirements for service composition development. Then we examine the information model and business rules required for the development of compositions. In Section 4 we describe the process of service composition development. Section 5 highlights related work and our contribution. We present our conclusions and future research in section 6.

2 Functional Requirements of Service Composition

When considering service compositions it is useful to identify two main use cases: service composition development and service composition management. During the process of service composition development, the application developer interacts with the service composition system to generate a business process by composing services. The use case starts when the developer sends a request. The system at the end produces an executable service composition.

In the second use case the application developer interacts with the service composition system to execute and manage compositions. This use case begins when the developer indicates that he wants to execute a service composition. In response the system gathers the required information and subsequently executes the composition. During run-time the developer may interact with the service composition system to make modifications, e.g. change service providers.

In this paper we only concentrate on the first use case, i.e., the development of compositions.

This paper advocates a phased approach to service composition development. The activities in this approach are collectively referred to as the *service composition life-cycle* [13]. Four broad phases are distinguished spanning composition *definition*, *scheduling*, *construction* and *execution*, as shown in Fig. 1.

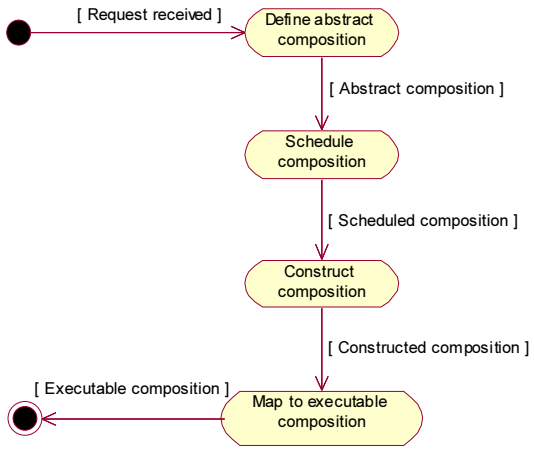

Fig. 1. Service composition life cycle

The idea behind the phased approach to service composition development is to start with an abstract definition and gradually make it concrete so that we can generate executable service processes from these abstract specifications.

The system starts in the *Definition phase* with an *abstract composite service*, which specifies the constituent activities of a composite service, the constraints under which they operate, their informational requirements, and the exceptional behavior that may occur during their execution

In the *Scheduling* phase of the approach, the service composition system determines how and when services should run and prepares them for execution. Its main purpose is to make the definition developed in the definition phase concrete by correlating messages to express data dependencies, and synchronizing and prioritizing the execution of the constituent activities. During this phase the system may generate alternative composition schedules and present them to the application developer for selection.

Next, the service composition system proceeds with the *Construction phase* to construct an unambiguous composition of concrete services out of a set of desirable or potentially available/matching constituent services. Similar to the

scheduling phase the system may produce alternative construction schemes (e.g. varying in quality or price) from which the application developer can select.

Lastly, during the *Execution phase* the service composition system prepares the constructed composed services for execution. This phase maps the resulting specification to an executable web service orchestration language (e.g. BPEL).

3 The Information Model for Service Composition Development

The information model (IM) is an abstract meta-model that represents the building blocks of all possible service compositions. The IM models the components required for a given composition as well as their inter-relationships. Relationships in the IM indicate how a composition is constructed. For example, a relationship between an activity and a flow indicates that this activity is used in the flow. We model all the required information as classes containing special purpose attributes so that this information can be captured and described. A service composition derived on the basis of the IM generates a specific instance of this model by populating its classes. The IM comprises classes referred to as *service composition classes*, while the instances of these classes are referred to *composition elements*. Relationships between composition classes, i.e., how to relate a certain activity to a flow, how to relate a service to an activity, and so on, are determined on the basis of business rules.

3.1 Service composition classes and elements

The IM is based on generic service composition constructs derived after a thorough study of the current standards (e.g. BPEL, BPML). Based on this study we have identified the following service composition classes: **activity, condition, event, flow, message, provider** and **role**. These classes and their inter-relationships are illustrated in Fig. 2, and presented in what follows.

– **Activity:** This abstract class represents a well-defined business function (similar to e.g. basic activities in BPML). It contains four attributes: `name, function, input, output`. An instance of this class can be defined as follows:

```
Activity: (
name="flightActivity"
function="flightTicketBooking"
inputs="departureDate,returnDate,from,to"
outputs="airline,flightNr,seatNr"
)
```

This example shows an activity named "flightActivity" that is meant for booking a flight ticket. It requires several input parameters to carry out this task, such as, for instance, departure and return date. The output parameters of this class include the airline name, and the flight and seat number.

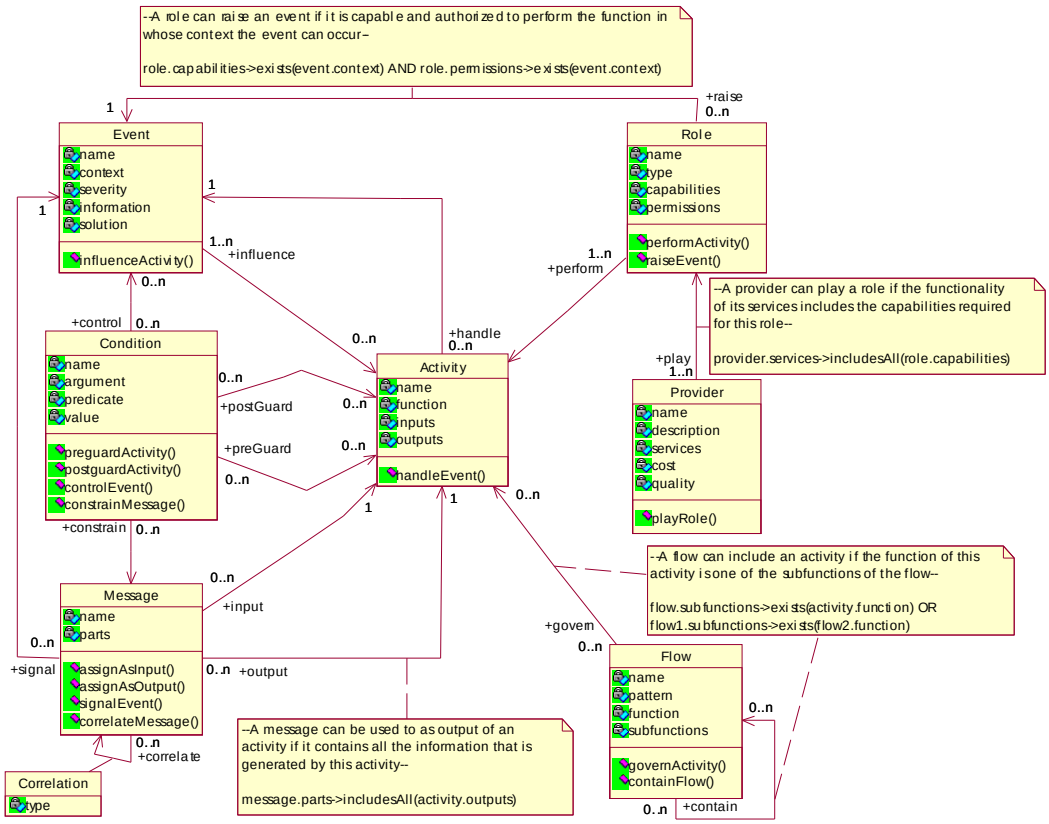

Fig. 2. Service composition model

– **Condition:** This class constrains the behavior of the composition by controlling event occurrences, guarding activities and enforcing pre-conditions, post-conditions and integrity constraints. To achieve this a condition class has four attributes, `name, argument, predicate, value`. A typical post-condition for "flightActivity" could be that "a seat has been reserved".
– **Event:** This abstract class describes occurrences during the process of service composition and its impact on an activity. These can be both of a normal and exceptional nature (e.g. encompassing WSDL faults). An instance of this class can be defined as follows:

```
Event: ( name="seatAvailabilityError"
context="flightTicketBooking"
severity="unrecoverable"
information="seatStatus"
solution="abandonCompositionExecution" )
```

This example illustrates an event class called "seatAvailabilityError". If the attribute "Severity" in this event class is set to "unrecoverable", then the

execution needs to be abandoned. To signal the occurrence of "seatAvailabilityError" a "seatStatus" message must be sent.

- **Flow:** This abstract class defines a block of activities and how they are connected. An example of an instantiation of the flow class can be:

```
Flow: (
name="TravelPlanFlow"
function="travelPlanning"
subfunctions="(flightTicketBooking,hotelRoomReservation)"
pattern="sequential"
)
```

  The above example shows a flow named "TravelPlanFlow whose function is "travelPlanning". Its subfunctions are "flightTicketBooking" and "hotelRoomReservation". These subfunctions are carried out in a "sequential" manner (indicated by the pattern attribute). Other patterns include "iterative", "parallel", "conditional", etc, as described in [13].
- **Message:** This abstract class represents a container of information (like e.g. properties in BPEL). Messages are used and generated by activities as input and output, respectively. They are also used to signal events, and can be correlated to other messages to express data dependencies. They have attributes such as name and parts.
- **Provider:** This abstract class describes a party offering concrete services, which can play a role(s) at runtime. A provider class declares attributes such as name, description, services, cost and quality. (Observe that no WSDL constructs are used here to describe providers to maximize the independency of the IM model with regard to particular standards)
- **Role:** This class provides an abstract description for a party participating in the service composition. Roles are responsible for performing activities and raising events. An instance of this class can be:

```
Role: (
name="flightRole"
type="airline"
capabilities="(flightTicketBooking,cancelTicketBooking)"
permissions="(flightTicketBooking)"
)
```

  The above example describes "flightRole" as being of the type "airline", both capable and authorized to book flight tickets.

Please observe that the above model closely resembles standard workflow meta-models, which have been developed (e.g. by the WfMC [12]). This is not surprising, since service composition is in many ways similar to workflow, e.g. concerning task structuring, transition conditions, roles, and etceteras. However, in the IM these are perceived and subsequently represented from a service oriented point of view. Also, some concepts like events are often not defined workflow meta models (e.g. in [12]), but they are an important part of the IM.

Now, at this stage it is easy to understand that the difference between an abstract, scheduled, and constructed service composition lies in the absence of specific composition elements or associations between these elements. More specifically, the service composition system starts by only specifying activities, messages and constraints elements in the abstract based on the user requirements

and leave the **flow, role, providers** elements unspecified. Then it can progress from an abstract service composition specification to an executable composition by gradually generating these elements on the basis of applying business rules and seeking user input. This is discussed in what follows.

3.2 Service composition rules

A concrete service composition needs to link elements such as "service provider" to "service", "service" to "activity", "activity" to "flow", and so on, as indicated in Fig. 2. These associations in a service composition IM are constrained by means of business rules. Fig. 2 shows some examples of these rules, referred to as *composition rules*, as notes attached to associations between service composition classes.

Composition rules are expressed in the Object Constraint Language (OCL) [9]). We apply such rules to constrain composition element attributes values and associations. An example of an attribute constraint is `activity.function="FlightTicketBooking"`, specifying that the function of an activity must be "FlightTicketBooking". The expression `activity.input -> notEmpty` is an example of an association constraint, depicting that the "input" of the activity must not be empty, i.e., an activity must always be associated with an input message.

Service composition comprises a number of composition rules that in our approach are classified into five broad categories of rules, namely structural, behavioral, data, resource and exception rules. These categories of business rules are discussed in what follows. To illustrate the concepts that we introduced we will use the composition elements illustrated in Table 1 to 7. In reality composition will likely be much harder as they may involve complex matching algorithms and conformance rules, which are not elaborated in this paper to provide more intuition.

Label	Name	Function	Inputs	Outputs
Activity1	flight	flightTicketBooking	departureDate,from,to	flightNr,seatNr
Activity2	hotel	hotelRoomReservation	checkinDate,duration,	hotelName
Activity3	car	carRental	period	pickupDate,carType
Activity4	stop	stopExecution	none	none

Table 1: Activity elements

Label	Name	Argument	Predicate	Value
Condition1	destinationCheckCondition	from	!=	to
Condition2	seatReservedCondition	seatNr	!=	-1
Condition3	seatUnavailableCondition	seatStatus	=	unsuccessful
Condition4	departureDateCondition	departureDate	>	currentDate

Table 2: Condition elements

Label	Name	Context	Severity	Information	Solution
Event1	seatUnavailableException	flightTicketBooking	unrecoverable	seatStatus	stopExecution

Table 3: Event elements

Label	Name	Function	Subfunctions	Pattern
Flow1	hotelCarFlow	hotelCar	hotelRoomReservation,carRental	ParallelWithSynchronization
Flow2	travelFlow	flightHotelCar	flightTicketBooking,hotelCar	Sequential

Table 4: Flow elements

Label	Name	Parts	Correlations
Message1	flightReservationData	departureDate,returnDate,from,to	
Message2	hotelRoomBookingData	checkinTime,duration,roomType	checkinTime=arriv-time
Message3	carRentalData	period,carType,insurance	
Message4	flightTicket	airline,dept-time,arriv-time,flightNr,seatNr	
Message5	hotelRoomConfirmation	hotelName,period,roomNr	
Message6	carRentalApproval	carType,pickupLoc,pickupDate,period,dropOffLoc	
Message7	seatUnavailableSignal	seatStatus	

Table 5: Message elements

Label	Name	Description	Services	Cost	Quality
Provider1	KLM	Royal Dutch Airline	flightSearching,flightTicketBooking	expensive	high
Provider2	MartinAir	Dutch Airline	flightSearching,flightTicketBooking	cheap	average
Provider3	Hertz	Car Rental Company	carRental	expensive	high
Provider4	Dollar	Car Rental Company2	carRental	average	average
Provider5	HotelDirect	Hotels Worldwide	hotelRoomReservation	cheap	average

Table 6: Provider elements

Label	Name	Type	Capabilities	Permissions
Role1	flightRole	airline	flightTicketBooking,bookingCancellation	flightTicketBooking
Role2	hotelRole	hotelBroker	hotelRoomReservation	hotelRoomReservation
Role3	carRole	carRentalCompany	carRental,carSale	carRental

Table 7: Role elements

Structural rules: rules in this category are used to guide the process of structuring, scheduling and prioritizing activities within a service composition. An example of a structural rule in this category can be defined as:

```
structuralActivity: flow.subfunctions->exists(activity.function)
OR flow1.subfunctions->exists(flow2.function)                    (1)
```

Suppose we have a travel plan composition consisting of flight ticket booking, hotel reservation and car rental activity, i.e., `Activity1-3` in Table1. One of the first concerns that a designer would face is to schedule these activities. We can observe from the structural rule (1) that we may only include an activity in a flow if its function attribute coincides with one of the subfunction attributes in a flow element. Consequently, `Activity1` in Table 1 can be included in `Flow2` in Table4. In a similar manner `Activity2` and `Activity3` can also be both included in `Flow1` (since the functions of `Activity2` and `Activity3` are subfunctions of `Flow1`). This yields the two flows: `Flow1` and `Flow2`. In order to merge these flows, we need to re-apply the structural rule (1). This time the rule indicates that we may include `Flow1` in `Flow2`. As a result we have a composition in which

the activities are scheduled in accordance with the specified dependencies and priorities, and the association instances are created by linking `Activity1` with `Flow2`, `Activity2-3` with `Flow1`, and `Flow1` with `Flow2`.

Data rules are used to control the use of data in a composition, i.e., how messages are related to each other, what is the necessary input/output message for an activity. There are four data rules: *assignAsInput, assignAsOutput, signalEvent* and *correlateMessage*. These are defined as follows:

```
assignAsInput:    message.parts->includesAll(activity.inputs)
assignAsOutput:   message.parts->includesAll(activity.outputs)
signalEvent:      message.parts->includesAll(event.information)
correlateMessage: message1.part != message2.part AND
                  message1.correlations->includes(message2.part)   (2)
```

To illustrate how the data rules *assignAsInput* and *assignAsOutput* are applied, we try to determine the input and output message for `Activity1` (Table 1). The rule *assignAsInput* indicates that a message is only suitable if it includes all the information required by the activity. `Activity1` requires an input message containing a departure date, a starting point and a destination. This is satisfied by `Message1` (Table 5) as it provides the required data. In a similar fashion we can derive the output message for `Activity1`. It must be noted that `Message1-3` are not suitable, since the outputs of `Activity1` are not all contained in one of these messages. However, `Message4` turns out to be suitable (even though it contains additional information).

We use the rule *signalEvent* to determine which message signals the occurrence of an event. To illustrate this, we consider as example `Event1` (Table 3), which describes a seat unavailability exception. The rule `Event1` also indicates that an appropriate seat status needs to be included in its signal. Thus the only message satisfying this requirement is `Message7`.

Lastly, messages may be correlated to express dependencies between data in a composition. The creation of such correlations is governed by a special rule called *correlateMessage*. This rule specifies that a correlation exists if a part attribute in a message is correlated to a part attribute in another message. For example, in `Message2` "checkinDate" must be equal to "arriv-time". Therefore `Message2` can be correlated with `Message4` by applying the rule *correlateMessage*.

Behavioral rules are used to derive conditions for guarding activities, controlling event occurrences and enforcing integrity constraints. We may define the following behavioral rules:

```
preguardActivity:  activity.inputs->exists(condition.argument)
postguardActivity: activity.outputs->exists(condition.argument)
controlEvent:      event.information->exists(condition.argument)
preserveIntegrity: message.parts->exists(condition.argument)   (3)
```

The first two rules derive the pre- and post-conditions of an activity, respectively. In particular, the rule *preguardActivity* specifies that a condition can

guard the execution of an activity only if its argument constrains an input of the activity. In other words, pre-execution guards can be based solely on information that is used as activity input. Take `Condition1` (in Table 2) for example, it constrains the inputs "from" and "to" of `Activity1` by stating that the starting place must not be equal to the destination. Therefore, `Condition1` can guard the execution of `Activity1` according to rule *preguardActivity*.

Another important use of conditions is to control how events can be raised by indicating in which situations they may occur. For this purpose the rule *controlEvent* is used to specify that if a condition constrains part of the information required to signal the event, then it can control the occurrence of that event. To understand this, we consider `Event1` (in Table 3) and `Condition3` (in Table 2). Rule `Event1` represents a seat unavailability exception, therefore it should only occur if no seats can be reserved. The rule `Condition3` can be used to constrain its occurrence, since this condition checks the value of "seatStatus" which is "information" required by `Event1` according to rule *controlEvent*.

Finally, conditions can be used to influence integrity constraints. For example, the rule *preserveIntegrity* guides the creation of constraints by establishing associations between **condition** and **message**. More specifically, rule *preserveIntegrity* specifies that only if the argument of the condition refers to a part in a given message, then the condition can be used to enforce data integrity. For instance `Condition4` (in Table2), which specifies a constraint that the departure date for the flight must always be greater than the current date, has an argument "departureDate". This argument is part of `Message1` (in Table5). Consequently, rule *preserveIntegrity* helps establish a valid association between `Condition4` and `Message1`.

Resource rules are provided to guide the use of resources in the composition in terms of selecting services, providers, and event raisers.

```
performActivity: role.capabilities->exists(activity.function)
                 AND role.permissions->exists(activity.function)
raiseEvent:      role.capabilities->exists(event.context)
                 AND role.permissions->exists(event.context)
playRole:        provider.services->includesAll(role.capabilities)(4)
```

The rule *performActivity* regulates which role is responsible for carrying out an activity in the composition. It indicates that this can only be the case whenever a role is capable and authorized to perform the function in the activity. For instance, in the case of `Activity1`, rule *performActivity* indicates that a role both capable of and authorized to book flight tickets needs to be found. As a result of this `Role1` (in Table 7) is selected to handle the functions of `Activity1`.

Roles are also responsible for raising events. The requirements for doing so are expressed by the rule *raiseEvent*. This rule specifies that a role must be capable of and authorized to perform the function in those contexts where the event can occur. For instance, only role `Role1` can raise event `Event1` according

to rule *raiseEvent*. This is due to the matching of the attribute "context" in `Event1` with the "capabilities" and "permissions" attributes in `Role1`.

At runtime roles are carried out by concrete service providers. To guide the selection process for each role we use the rule *playRole*. This rule controls selection by demanding that a provider's services must provide the functions for which the role is capable of and authorized to perform. This means that, for example, `Role1` requires that a service provider must offer a service with flight ticket booking. According to this rule, *playRole* `Provider1` (Table 6) is the first suitable provider.

Exception rules are finally used to guide the exceptional behavior regarding service compositions. In this case the *influenceActivity* is used to determine which events can affect an activity at run-time, while the rule *handleEvent* governs how these events are to be handled. These rules are defined as follows:

```
influenceActivity: event.context->includes(activity.function)
handleEvent:       activity.function=event.solution          (5)
```

Rule *influenceActivity* specifies that for an event to impact an activity at runtime, the context in which the event occurs must be equal to the function of the activity. According to this rule an event such as `Event1` can only influence an activity whose function is the attribute "flight-ticket-booking". As a result, the only element with which an association can be established is `Activity1`.

Knowing which activity is affected by which event is relatively useless if it is not clear how this event is handled. Each event specifies in its "solution" attribute the preferable way to react to its occurrence. The rule *handleEvent* is used for this purpose by specifying that an activity can only handle an event if its "function" is equal to specified "solution" in the respective event. For `Event1` this means finding an activity whose function is to "stop execution" of the composition. Accordingly, `Activity4` is assigned to handle `Event1`.

4 Service Composition Development Process

In this section we use the constructs we introduced in the previous to show how to construct service compositions. We assume that already defined composition elements, such as the ones described in Table-1, as well as all composition elements in Tables 2 to 7, are stored in the repository of the service composition tool (see Fig 3). We also assume that the user is interested in booking a flight from New York to Vancouver with departure date July 15th, and return date August 22th. Furthermore, the user needs to reserve a hotel room and rent a car.

The process of designing a composite service for this travel example becomes a matter of applying the composition rules to incrementally construct composition elements and associations. Again we use the composition elements in Table

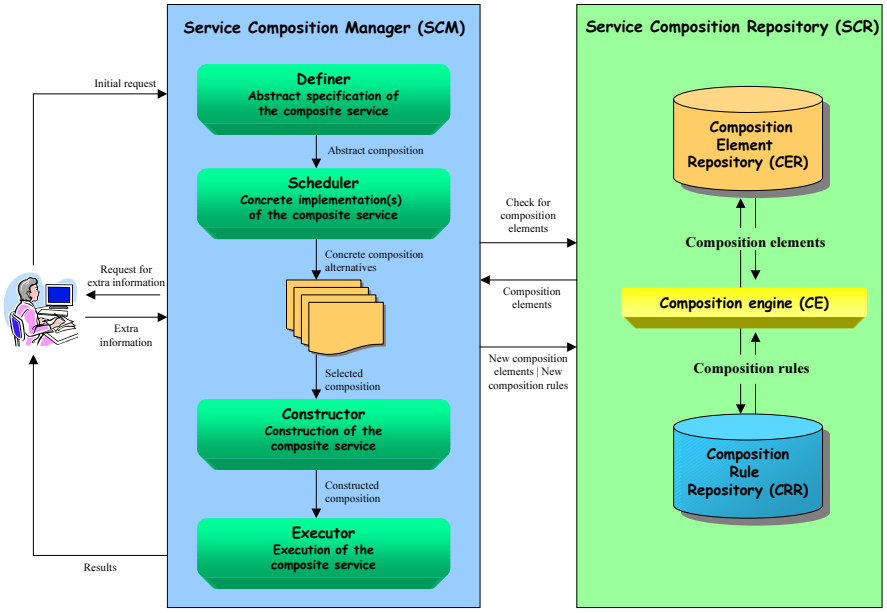

Fig. 3. Architecture for the Service Composition Development System

1 to 7 for this example. The user triggers a request by either writing an application program that retrieves existing activity elements and tries to combine them in a composite service or by issuing a request expressed in a formal XML based request language such as the one described in [1], [10]. The process of constructing composite services out of elementary activities can proceed in accordance with the steps of the algorithm found in Fig. 4, and is described in the following. To describe this process we will use the travel example we introduced in the previous.

The service composition development system receives the user request and enters the *Definition* phase of the algorithm depicted in Fig. 4. In the first instance the system attempts to *determine/select activities* that satisfy a user request. As a result of the request expressed above, `Activity1, Activity2` and `Activity3` (in Table 2) are added to the composition. Subsequently, for each activity the service composition development system tries to *add message exchanging behavior*. To achieve this the system must determine for each activity what type of messages it should use as input and generate as its output messages. For example, for `Activity1` message `Message1` is found as input and `Message4` as output. As a result these elements are added to the composition.

The next steps in the algorithm *defines exceptional behavior* for the service composition by applying the rule `influenceActivity` for events and activities possible to determine possible event occurrences. For example, for `Activity1`

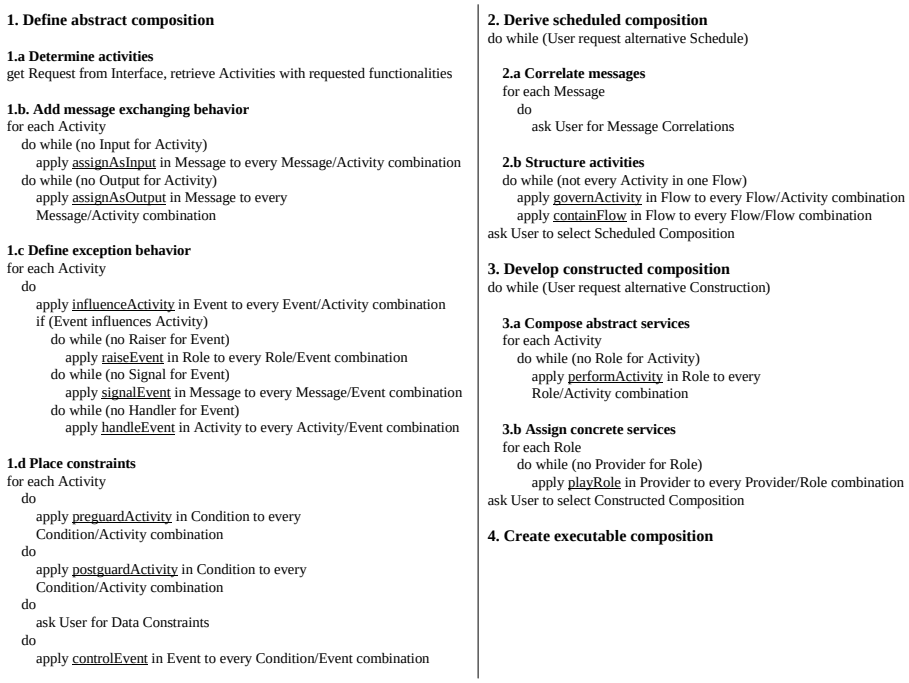

Fig. 4. Algorithm for the Service Composition Development Process

`Event1` (Table 4) is found as an influencing event. If an event influences an activity, subsequently the event raiser, signal and handler need to be determined. For `Event1` this is done as follows:

1. Apply `raiseEvent` to **role** to determine the role raising the event. For `Event1` `Role1` (Table 8), is found.
2. Apply `signalEvent` to **message** to derive the message signalling the event occurrence. As a result `Message7` is found for `Event1`.
3. Apply `handleEvent` to **activity** to determine the activity that will handle the event. For example `Activity4` is found for `Event1`.

Following the definition of exceptions, necessary constraints (if any) are placed on the service composition. The constraints under which the composition is to be executed are depicted in condition elements. These can be predefined, e.g., pre-condition, post-condition, or user specified, e.g., data constraints. As an example we can derive pre-conditions for the activities in the composition by

applying `preguardActivity` for each activity. This means that for `Activity1` condition `Condition1` applies.

Next, the abstract composition is made more concrete by entering the *Scheduling* phase in the algorithm. During this phase we need to *correlate messages* and *structure activities*. Correlations are usually context-dependent and thus cannot be derived by a general business rule. Instead they can be defined for each message by the user. This may, for instance, mean that for the travel request we consider, the user may wish to define a correlation between "arriv-time" in `Message4` and "checkinDate" in `Message2` to ensure that he will have a hotel room the day he arrives. Following this, activities must be structured. This is accomplished by applying the `governActivity` and `containFlow` structuring operations which group related activities into a single flow. The construct `governActivity` can be applied to both `Activity2` and `Activity 3` (Table 1) to indicate that they can be included in `Flow1` (Table 5). It can also be applied to `Activity1` to include it in `Flow2`. Finally, the constructs `Flow1 Flow2` can be combined into a single flow using `containFlow` to create a complete activity schedule. The correlation and structuring sub-steps may be repeated to generate additional schedules that may be relevant to a user request.

During the next step the algorithm enters *Construction* phase during which the scheduled composition will turn into an unambiguous composition of concrete services. First the algorithm *composes abstract services* by associating each activity with a role, specifying the requirements for a party interested in carrying out the activity. This is accomplished by applying the operation `performActivity` for each activity until a role has been found. For example, for `Role2` can be found for `Activity2`. As a last sub-step in the construction phase *concrete services are selected* for the roles in the composition. For this purpose the operation `playRole` is applied to each role, resulting in, for example, `Provider2` (Table 6) as the first suitable provider for `Role2`. The construction sub-steps can be repeated to create multiple concrete compositions for the user to choose from.

The final step in the algorithm is the *Execution* phase during which the constructed composition is mapped to an executable format in a service composition language, e.g., BPEL. Such a translation can be likely done without too much difficulty, however, we do not elaborate on it here due to space limitations.

It is not necessary that every composition needs to go through each individual step discussed in the previous. If, for example, part of a composition is already partially constructed with some of the composition elements defined, the model only needs to be completed and mapped to an executable format.

5 Related Work

Most of the work in service composition has focused on using work flows either as a engine for distributed activity coordination or as a tool to model and define service composition. Representative work is described in [3] where the authors

discuss the development of a platform specifying and enacting composite services in the context of a workflow engine.

The workflow community has recently paid attention to configurable or extensible workflow systems which present some overlaps with the ideas reported in the above. For example, work on flexible workflows has focused on dynamic process modification [8]. In this publication workflow changes are specified by transformation rules composed of a source schema, a destination schema and of conditions. The workflow system checks for parts of the process that are isomorphic with the source schema and replaces them with the destination schema for all instances for which the conditions are satisfied.

The approach described in [6] allows for automatic process adaptation. The authors present a workflow model that contains a placeholder activity, which is an abstract activity replaced at run-time with a concrete activity type. This concrete activity must have the same input and output parameter types as those defined as part of the placeholder. In addition, the model allows to specify a selection policy to indicate which activity should be executed.

In [14] authors developed an agent-based cross-enterprize Workflow Management System (WFMS) which can integrate business processes on user's demand. Based on users' requirements, the integration agent contacts the discovery agent to locate appropriate service agents, then negotiates with the service agents about task executions. Authors in [15] proposed a dynamic workflow system that is capable of dynamic composition and modification of running workflows by using a business rule inference engine. However these two approaches are more of the focus of dynamic process execution and management.

Our approach differs from the above work as regards supporting the dynamic composition of web services in the following manner:

- We propose a model driven approach towards service composition, which covers the entire service composition life cycle ranging from abstract service definition, scheduling, construction, execution and evolution. By raising the level of abstraction compositions developed in our approach are flexible and agile in the face of change.
- Service compositions are defined in terms of basic abstract elements, i.e. *composition elements*, which are used to construct a concrete service composition specification. This design process is governed by *composition rules*, supporting highly flexible composition development.
- Business rules are classified based on the requirements of service composition, something which to the best of our knowledge has not been addressed in work related to business rule classification, such as [5],[7] and [11].

6 Conclusions and future research

Current standards in service composition, such as BPEL, are not suitable for dealing with the complex and dynamic nature of developing and managing com-

posite web services to realize business processes. With a vast service space to search, a variety of services to compare and match, and different ways to construct composed services, manual specification of compositions is an almost impossible task requiring specialistic knowledge, taking up much time and effort. The challenge is thus to provide a solution in which dynamic service composition development and management is facilitated in an automated fashion.

In this paper we have presented a phased approach to service composition development conducted on the basis of abstract constructs provided by a model driven architecture. Service compositions are constructed in a piecemeal fashion by progressing from abstract service descriptions to more concrete ones on the basis of a set of business rules that synthesize the activities in a composition. This approach makes service composition more flexible and dynamic compared to current standards and recent research activities.

The work presented herein is at an initial stage. Several issues including the mappings and conformance between compositions need to be further investigated and verified in a formal manner. In addition, a change management sub-system to control the evolution of business rules and service composition specifications needs to be developed.

References

1. Aiello et al. A request language for web-services based on planning and constraint satisfaction. VLDB Workshop on Technologies for E-Services (TES02), 2002.
2. Business Process Modelling Initiative. Business Process Modeling Language, *June 24, 2002, http://www.bpmi.org*
3. F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, M.C. Shan. Adaptive and Dynamic Service Composition in eFlow, *HP Lab. Techn. Report, HPL-2000-39.*
4. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, S. Weerawarana. Business Process Execution Language for Web Services, *July 31, 2002, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/*
5. Business Rules Group. Defining business rules, what are they really?, *July 2000, http://www.brcommunity.com*
6. D. Georgakopoulos, H. Schuster, D. Baker, and A. Cichocki. Managing Escalation of Collaboration Processes in Crisis Mitigation Situations. *Proceedings of ICDE 2000, San Diego, CA, USA, 2000.*
7. B. von Halle. Business rules applied: Building Better Systems Using the Business Rule Approach, *Wiley & Sons, 2002*
8. G. Joeris and O. Herzog. Managing Evolving Workflow Specifications with Schema Versioning and Migration Rules. *TZI Technical Report 15, University of Bremen, 1999.*
9. Object Management Group. Object Constraint Language, *http://www.omg.org/docs/formal/03-03-13.pdf*
10. M.P. Papazoglou, M. Aiello, M. Pistore, J. Yang Planning for Requests against web-Services *IEEE Data Engineering Bulletin*, vol. 25, no.4, 2002.
11. R. Veryard. Rule Based Development, *CBDi Journal, July/August 2002*
12. Workflow Management Coalition. The Workflow Reference Model, *http://www.wfmc.org/standards/docs/tc003v11.pdf*
13. J. Yang, M.P. Papazoglou. Service Component for Managing Service Composition Life-Cycle, *Information Systems, Elsevier, June, 2003*
14. Liangzhao Zeng, Boualem Benatallah, and Anne H. H. Ngu. "On Demand Business-to-Business Integration", *CooPIS01, Trento, 2001*
15. Liangzhao Zeng, David Flaxer, Henry Chang, Jun-Jang Jeng. PLM$_{flow}$-Dynamic Business Process Composition and Execution by Rule Inference, *TES2002, Hong Kong, 2002*