Domain Objects for Dynamic and Incremental Service Composition

Antonio Bucchiarone, Martina De Sanctis, Marco Pistore

Domain Objects for Dynamic and Incremental Service Composition

Antonio Bucchiarone, Martina De Sanctis, Marco Pistore

Fondazione Bruno Kessler, Via Sommarive, 18, Trento, Italy
{bucchiarone,msanctis,pistore}@fbk.eu

Abstract. A key feature of service-based applications (SBAs) is the capacity to dynamically define the composition of services independently available, which is required to achieve user goals. For this reason, to effectively deal with the obstacles due to continuous context changes, an incremental refinement of provided services is needed. We propose a new model that allows service functionalities to be defined partially, through the use of abstract activities. The refinement of these activities is postponed and performed incrementally at runtime, using the actual context as a guide. Our approach lets a service provider avoid the hard-coding of all service functionalities and their possible compositions at design time, delaying their refinement until the execution phase. Consequently the whole SBA's design becomes modular and flexible to better meet the typical dynamism of SBA. We illustrate the approach through an example scenario from the urban mobility domain.

1 Introduction

The high dynamism of the environments in which *service-based applications* must operate, together with their context continuously changing, make the deployment and maintenance of complex distributed applications a hard task to accomplish in a really efficient way. Services may enter or leave the system at any time. Moreover, the situation in which they are executed may be different each time or it may change during their execution. Besides the end-users may change their preferences and emergent requirements can arise. In many situation it often occurs that the only way the application can manage such changes is at run-time, since current situations, available services and users needs are not known priori. *Incremental and dynamic service composition*, hence, becomes the key point to address the problem of continuously changing environment.

The existing approaches of service composition [19], have some crucial limitations. Most of them are based on the assumption that during the composition requirements specification, the application designer knows the services to be composed. Consequently, it often occurs that composition requirements include specific implementation details of the services with which they are supposed to be used. Thus, requirements are strongly linked to particular service implementations and they cannot further be used with similar but different services. However, in dynamic systems, both the composition requirements and the available services change frequently. For these reasons, a static specification of composition requirements upon services is not adequate. In this setting, it is necessary to be able to produce service compositions consistent with the surrounding context and to quickly manage emerging user requirements during the services execution [20]. To fulfill this purpose, a service model reflecting the contextual environment of services has to be provided. Moreover, the model has to be flexible enough to cope with the typical dynamism of service-based applications. This objective can be reached by avoiding the need of hard-coding every features of the context and/or services at design time [3], but rather leaving the application to dynamically discover the context around and, thus, to incrementally make service compositions.

In this paper we propose an approach, the unified service description "domain objects", that allows a partial definition of such services in order to enable their incremental composition when the context is discovered or when context changes are detected. In addition, the idea is to allow autonomous, heterogeneous and distributed services to be presented in a standard, open and uniform way. The incremental composition task is made by defining *abstract activities*, which are activities that have not a concrete implementation but they are defined

in terms of a *goal* to be reached, in the context space. They represent the *opening points* in the application design in which the incremental refinement process can take place during the run-time phase. As concern the management of the dynamism of SBAs, the domain object model allows the designer to model a hierarchically structured conceptual network of services, which are modeled in a modular way that guarantees an efficient management of the continuous entrance and exit of services in the system, avoiding the need of re-designing of the application. The modularity feature is also given by the concept of *fragment* that is used to represent the protocols which have to be performed to execute the services, in a customizable and portable way. Fragments play a key role also in the process of the incremental composition of services because they are used to replace the abstract activities, by making them able to reach their goal using the service available in the context.

The paper is organized as follows: we start with a motivating scenario to understand the requirements that a service-based application must satisfy, in Section 2. Section 3 describes the general approach, firstly, in Section 3.1, by defining how the services are modeled using our approach; then, in Section 3.2, by explaining how to design a service-based application using the domain objects model; and, finally, by clarifying how the bottom-up approach for the incremental and dynamic refinement process is realized, in Section 3.3. Section 4 is devoted to the definition of the formal framework, followed by the implementation discussion of a prototype in Section 5. Related work are discussed in Section 6. Finally, our conclusions and directions for future work are presented in Section 7.

2 Motivating Scenario

In last years, the concept of smart-cities is increasingly catching on. In this context, in which each city is assumed as a system of systems, we focus on the *urban mobility domain*. The urban mobility system is a network of multi-modal transport systems (e.g., buses, trains), services (e.g., car sharing, bike sharing, car-pooling) and smart technologies (e.g., sensors for parking availability, smart traffic lights) strongly interconnected to better manage mobility by offering smart services. Today, also modern transportation services are increasingly prevalent, such as the *Flexi-bus* service that combines the features of the taxi and the regular bus service. A Flexi-bus system defines a network of pickup points and provides passengers with a way to get around between any of these points, on demand. In other words, a passenger can request a transit between any two points at a given time. The basic idea of the service is to organize bus routes in such a way that all requests are served with minimal number of buses/routes. A urban mobility application is made of autonomous and heterogeneous services, which are offered by different service providers, that can be composed in order to achieve specific user needs. The providers are *autonomous* from each other in the system, as they can take decisions in an independent and distributed manner. Each service may enter or leave the system at any time, as well as it may change its offered functionalities or it may offer new ones, making the system open and dynamic. The dynamism is also given by the system's context, whose change can affect the operation of the system (e.g., traffic jams, bus delays, on-line payment services unavailable, strikes). During the system execution, all these aspects may generate new properties when combined together, maybe bringing to different mobility solutions because of the enabling or disabling of services. However these informations are only known at runtime, therefore it is no possible to predict every potential solution that can be offered, in terms of services composition, because of the incomplete knowledge at design-time. A urban mobility application is essentially made of *entities*, which provide services, and *relations* between them. The model of the scenario is based on the idea of an extended service model, properly hosted by SCA [18]. SCA comes with built-in extensibility capabilities. The SCA assembly model is defined such that extensions can be added for new interface types or for new implementation types, or new binding types. However, the definition of extensions is left to the implementor. This is witnessed by different SCA implementations (e.g., Apache Tuscany [13]), but many of these specialties are not available at programming level. The *entities* can be organized in two main categories: (i) the **service consumers**, which comprises the *end users* that daily make use of mobility services, *private or public companies* that might want to create value-added services (VASs) by exploiting the urban mobility environment's services; (ii) the **value-added service (VAS) providers** that, by exploiting and composing more basic

services or VASs available in the environment, create new VASs to be forwarded in the system. A VAS provider can play both roles of consumer, if it does not forward the new created service in the system, and provider, if it does, as we will see later. As regard *services*, we already said that we can have basic services available in the domain (e.g., smart traffic lights, flexi-bus service, train service, on-line payment service, parking service) or VASs (e.g., *route planner* able to provide a list of flexi-buses optimized routes, calculated by exploiting some basic services in the environment, such as those giving information on the current situation of traffic, pollution and weather together with the current number of requests coming from users). The different entities must be able to *collaborate* in the creation of services optimizing the resources or the quality of the system (e.g., reduction in traffic and emissions of CO2). Moreover, there are the mobility solutions eventually provided to the requesting entities, possibly coming from the composition of different services. These must be customized on the needs expressed by requesters (e.g., user preferences and profile must be taken into account when choosing the services to compose). The need for customization also implies the need of *adaptive* services, capable of adapting their behavior dynamically. All the entities must interact together, each with different roles and purposes. As we said before, being the context continuously changing, these interactions must be *dynamic* and *context-aware* with respect to the surrounding environment (e.g., the application must be able to detect new available services or the unavailability of existing ones). In conclusion, a urban mobility application must meet specific requirements, as revealed by the scenario, such as: (i) *dynamism* to manage continuous changes; (ii) *openness* to address the problem of the services that can enter or leave the system at any time; (iii) *autonomy of the entities* to reflect the independence of the providers and their services; (iv) *context-awareness* to consider the availability/unavailability of services during the composition phases and the users profiles and preferences; (v) *adaptivity of services* to reflect the dynamic behavior of involved services; (vi) *collectivity* to allow entities to collaborate for realizing optimized services; (vii) *customization* to offer services which are not general and statically defined but services customized for each user on the basis of their specific needs. Those requirements are fundamental to really fulfill the main features of a modern and dynamic urban mobility application.

3 General Approach

In this Section, we discuss an approach at a conceptual level, by introducing all the main concepts through the exploration of the scenario depicted in Figure 2. In Section 3.1, we explain how the entities in the environment are modeled; in Section 3.2 , we explain how to design a service based application using *domain objects*, while, in Section 3.3 we define how the services are incrementally refined and composed during the execution phase.

3.1 Entity Representation

Firstly,the approach is designed around the concept of *domain object* (DO). The DOs are used to model the entities, both humans and systems (e.g., users, service providers), with their features and their behavior, in a *standard*, *open*, and *uniform* way. To describe the model of a DO, we refer to the *FlexiBus Manager (FBM)* DO, in Figure 1, which comes from to the *Urban Mobility Application (UMA)* scenario depicted in Figure 2 and whose design is illustrated in the next Section. A DO is represented as a model made of two layers, namely the *core layer* and the *fragments layer* (see Figure 1). Briefly, the business of the FBM mainly consists in the management of the flexi-bus service, the definition of optimal routes for flexi-buses and the management of requests for on line ticket payments. The core layer defines the *structure*, the *interface* and the internal *behavior* of a DO. The structure represents the state of the DO and it is made of:

- a set of *variables*, which represents its features. For example the state of the FBM is made of the optimized *routes* dynamically defined, the information about the *flexi-buses* in action in the city, the *TicketStatus* related to the payment of the ticket by a user for the booked Flexi-bus, etc..

– a set of *relations* to model domain objects' direct connections. For example, the FBM holds relations with the instances representing the real flexi-buses running around the city, as shown by the *fbInstances* relation.

The interface, as depicted in the right side of the core layer, consists in:

– a set of *subscriptions definition* that associates reaction functions to some events coming from other DOs. As an example, the *Traffic monitor* subscription in the FBM triggers the execution of the *ManageTrafficInfo* process by reacting to a *Traffic Jam Notification* forwarded by the Traffic Management Service (TMS). This subscription is shown by the arrow labeled with (a).
– a set of *ports* that define *custom events* that a domain object may generate. They carry information and represent changes in the domain object's structure. As an example, the FBM publishes on the *routes update* port to notify that new routes are available. It is then possible that other DOs subscribe to this port to be triggered in case of the availability of new routes. This subscription is represented by the arrow labeled with (b), which shows that the UMA has made a subscription to the *routes update* port of the FBM.
– a set of *service notifications* that are used to propagate events and/or updates from a service, without an explicit request. This is realized by publishing the events on specific ports, as shown by the dashed arrows in the Figure. Examples of service notification are the *FlexiBus service unavailable* notification forwarded by the FBM and also the *Traffic Jam Notification* of the TMS.

The behavior of a DO represents all the processes that it implements to execute its services. A process is represented as a sequence of activities, also complex with loop and/or conditional steps. The FBM, for example, has three main processes such as *RoutePlanner*, *PaymentRequestManagement* and *FlexiBusBooking*.

In the fragments layer, the set of services that the DO externally exposes are modeled. Each service is represented as a *fragment* [8] that represents the interface with an internal process in a DO. The FBM for example, provides two main fragments, namely the *BookFlexiBus* and the *TicketPaymentRequest*. As regards activities, they can be essentially of four types: *input*, *output*, *concrete* and *abstract*. While the first three are well known, the novelty is the use of the *abstract activities* to make fragments, and thus services, dynamic. An abstract activity is defined in terms of the *goal* it needs to achieve. The goal consists in a configuration of the state of the DO holding the abstract activity that has to be reached. For example, the *TicketPaymentRequest* fragment ends with the abstract activity *Pay*, which is drawn with a dashed border and which has the goal "*TicketStatus = paid*". The TicketStatus is a variable in the state of the FBM and its initial value is *notPaid*. Each non abstract activity can be executed both autonomously by the fragment or by interacting with the processes in the core layer, through direct communication. For example, considering the *BookFlexiBus* fragment of the FBM, the activity called *Choose Route* performs its task with no interaction with the core layer. To the contrary, its output activity *Send Request* is aimed to trigger the internal process named *RoutePlanner*, by calling its input activity *Receive Travel Request*. In the Figure, this kind of connection between activities is identified by tagging them with the same label. Abstract activities, instead, stand for tasks whose implementation is not known a priori and must be produced at run-time through the composition of fragments (so-called activity refinement) [4]. The execution of these fragments leads the process to reach the goal defining the abstract activity. As regards interactions and cooperations between DOs, these can be realized in two ways:

1. since the interface of the core layer of a DO is public, a direct connection can be established by using the mechanism of subscriptions. Example of these connections are expressed by the arrow labeled with (a) and (b) in Figure 1. These connections are defined during the design phase of an application. They allow the engineer to construct a *hierarchically structured conceptual network* of domain objects, having possibly different levels of abstraction. In such a hierarchy, usually the DOs in a layer can *monitor* the DOs in the layers below. The connections highlighted in Figure 1 by the solid arrows define the hierarchy made of the Urban Mobility Application which monitors the FBM which monitors the TMS;

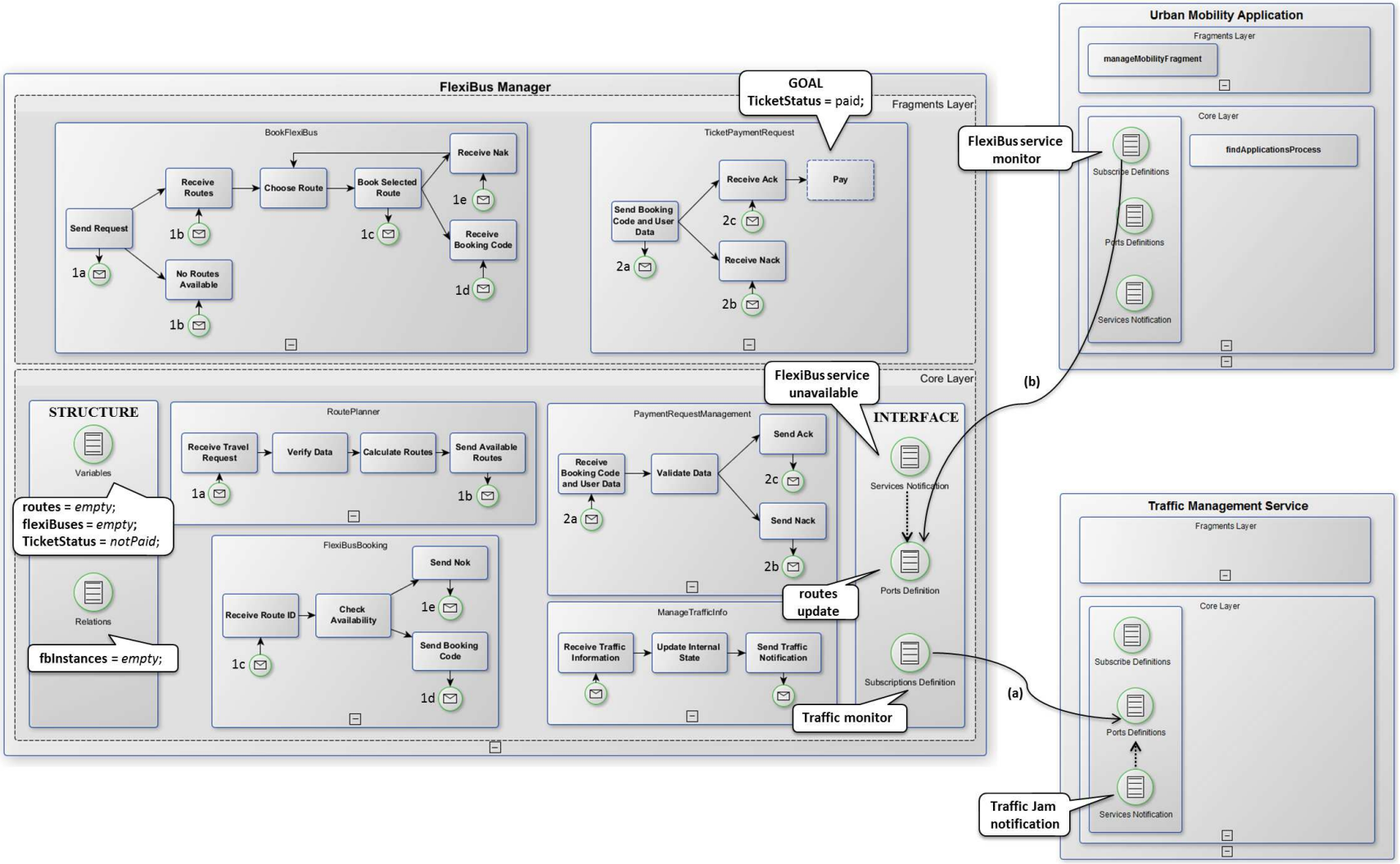

Fig. 1: FlexiBus Manager Domain Object.

2. a second way consists in collaborations between DOs realized dynamically at runtime, through the exchange of fragments during the execution of abstract activities. In this case, the interacting DOs are not directly connected by explicit relations, but they interact during the runtime phase. For example, the *Pay* abstract activity of the *TicketPaymentRequest* fragment will be refined by using one ore more fragments of other DOs, as we will in Section 3.3.

3.2 Service Based Application Design

In this Section we present our approach to design a SBA through the definition of correlations and cooperations between a multitude of DO. Moreover we present how domain objects are able to refine incrementally their services, while executing them. The scenario depicted in Figure 2 drives us through the design of the application by showing how all the entities are modeled, how service providers are designed, how a domain objects hierarchy is build up using relations between DOs and, finally, how mobility solutions are provided to the end-users through the runtime selection and composition of different fragments coming from the involved domain objects. At the bottom of Figure 2, all the entities that may contribute to

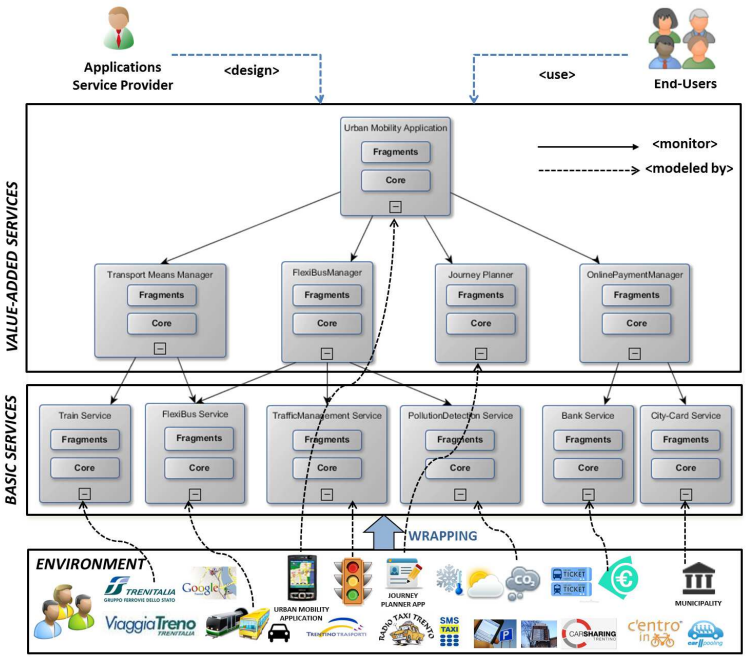

Fig. 2: Urban Mobility Application Scenario.

the overall application are shown. They frame the environment in which the application lives. It is composed by a multitude of entities, both humans and systems, which are autonomous and disconnected from each other. In the case of systems, the interaction exploits all the IT interfaces these systems expose on the web. These IT interfaces are however fragmented and heterogeneous, i.e., they consist of software designed independently from each other, and they are made available through a large variety of different technologies (web pages, web APIs, REST or SOAP services, feeds, open data, and so on). There is hence a need to encapsulate interactions with systems, so that they are presented in a standardized way. In our approach, this is achieved by a specific *wrapping* layer, which has the goal to cope with the encapsulation of fragmented and heterogeneous sources, aiming at presenting them as open, uniform and reliable *services*. In this scenario, there are people that make use of services by using applications on their mobile devices (e.g., journey planner app); many mobility service

providers (e.g., flexi-bus, car pooling, parking service); municipalities that can offer smart services for citizens (e.g., the *city-card* that is a smart rechargeable card enabling discount and facilities to the owner); traffic and security management systems; pollution detection systems; weather forecast systems; systems for geo-location; banks and other public or private companies that can be involved in a urban scenario (e.g., banks offering on line payment facilities). For better comprehensibility, in our design, we show only the domain objects that are involved in the service composition.

Basic Services. The layer just above the environment shows the domain objects modeling the basic service providers. The flexi-bus service, just notify real-time information on the instances of flexi-buses all around (e.g., their position, their delay if any). While, the train service also offers a fragment, namely *trainBooking*, allowing the user to book a train and possibly to pay the ticket on line. The *Traffic Management Service* and the *Pollution Detection Service* domain objects essentially release information of real time traffic and pollution situations respectively. Finally, the *Bank Service* and the *City-card service* domain objects expose different fragments, all related to the management of the on line tickets payment and connected services. In detail, the bank service has one main fragment, namely the *onLine-Payment*, while the city card service offers two fragments, the *payTransportMeansTicket* and the *RechargeCityCardOnLine*.

Value Added Services. Going up in the design of the scenario, there are the most interesting layers, in theDOs hierarchy containing the DOs modeling the *VAS providers*. These are the *Transport Means Manager*, the *Flexi-bus Manager*, the *Journey Planner* and the *Online payment Manager* in the middle layer, and the *Urban Mobility Application* in the top layer. These providers are characterized by the ability to offer value added services, in two possible ways:

1. by monitoring two or more basic service providers and then computing new services by exploiting them. For example the Transport Means Manager in Figure 2 monitors the DOs modeling the Train Service and the FlexiBus Service. In Figure 2, this kind of relation is shown by the solid arrows, which are marked with the <monitor >label.
2. by defining abstract activities in their fragments and/or processes that can be dynamically refined through the composition of others fragments currently available in the application.

The FBM introduced in Section 3.1 is an example of a VAS provider. Its business, depicted in Figure 1, consists in the management of the flexi-bus service by monitoring all the flexi-bus instances and in the definition of optimal routes for flexi-buses, which are calculated by considering the current situation of traffic and pollution. The routes are provided by the *RoutePlanner* process in the core layer. It is a clear example of VAS, which is realized by exploiting the services offered by the *FlexiBus Service*, the *Traffic Management Service* and the *Pollution Detection Service* in the layer below. The FBM also handles the requests for the on-line payment of the tickets for flexi-buses. It exposes two fragments:

1. the *bookFlexiBus* that corresponds to the protocol that must be performed by a user to book a flexi-bus. It works interacting with the processes in the core layer, as we said in the previous Section.
2. the *ticketPaymentRequest* that, by receiving the booking code of the chosen flexi-bus and the user data, triggers the internal process *PaymentRequestManagement* to verify if the user is allowed for the on line ticket payment. If so, the fragment execution can continue until the abstract activity *Pay*. The SBA will refine this abstract activity.

3.3 Incremental Service Composition

In this paper, starting from the domain object model proposed in [5], we extend it by introducing the concept of *fragment*, coming from [4], with its flexibility characteristic in modeling modular and dynamic services that can be easily composed. In particular, in this work we focus on the need for refining an abstract activity within a fragment/process instance. In our approach we model the incremental refinement process of an abstract activity by assuming that the adaptation engine [21] provides to the application the fragments composition on the basis

of the goal of the abstract activity. In the majority of the approaches of service composition, *top-down* techniques are used. Essentially, a service composition is first defined and then deployed for being executed. This is not useful in a dynamic environment where services can enter or leave the system in any moment while end-users are constantly moving around, discovering new services and changing their needs. Our approach follows a *bottom-up* procedure. The fragments selected for the composition replace the abstract activities and are executed. To explain the approach, we present and example from our running scenario. Suppose that there are two end-users, namely Marco and Paolo, that use the urban mobility application. Its main fragment, namely the *manageMobility* shown in Figure 3, consists in forwarding travel requests to its internal process that, after exploiting the monitored services, will send back the available applications in the environment that are able to manage multi-modal transport services.

Fig. 3: manageMobility Fragment.

The *journey planner* (JP) is a mobile application aimed at supporting sustainable urban mobility by offering multi-modal travel solutions. The JP allows users to send travel requests, by indicating different preferences (e.g., starting and destination's points, departure hour, preferred mode of transportation) and personal profile. After validating the request, the application is able to provide multi-modal transport solution(s) from which the user can possibly choose. Marco has both the city-card and the bank account while Paolo has not the city-card. In addition, Marco expresses the preference of moving by using a flexi-bus while Paolo prefers to go by train. We show how, following the bottom-up approach, two different final mobility solutions, customized for two different requests and user profiles, are incrementally defined. The urban mobility system receives the requests from Marco and Paolo. It replies by suggesting the usage of the JP application. The JP receives the two forwarded requests and it creates two customized transport solutions. As regards Marco, the incremental refinement process is shown

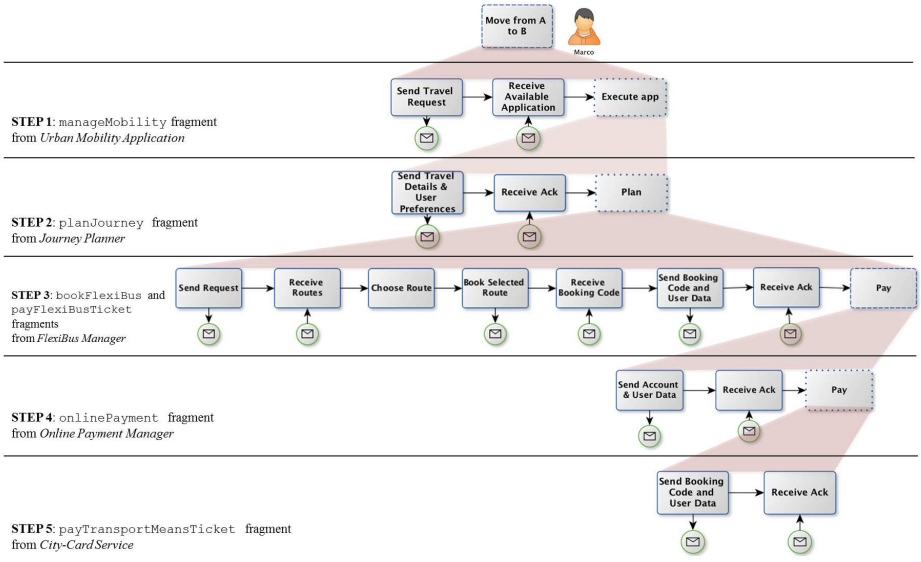

Fig. 4: Incremental Service Composition.

in Figure 4. The process starts with the need of Marco of moving from a point A to a point B. This need corresponds to an abstract activity of the user DO. The *manageMobility* fragment of the UMA is executed. It provides the JP application, as response. Thus, in the second step, the *Plan* abstract activity of the *planJourney* fragment of the JP has to be refined. Now,

knowing that Marco prefers to go by flexi-bus, the selected fragments for the third step are the *bookFlexiBus* and the *TicketPaymentRequest* of the FBM, which are composed in sequence, giving the protocol to execute. But the *Pay* activity is abstract so, in the step number four, the fragment *onLinePayment* of the Online Payment Manager is provided to start the payment procedure. At this point is not yet known the real protocol to be performed for paying. In fact the fragment ends with an abstract activity also called *Pay*. As Marco has expressed the wish of paying with his city-card, the last provided fragment is the *payTransportMeansTicket* of the CityCard Service, which is necessary, finally, to execute the on-line ticket payment. Through five steps of refinement, the final protocol for the whole service execution has been provided in a dynamic and context-aware manner. As concern Paolo, instead, the service composition is almost equal to the one made for Marco but there are two important differences, exactly in the third and the fifth steps of the process of refinement. At the third phase, the provided fragment is the one exposed by the *Train Service* DO, namely the *trainBooking*, as shown in Figure 5(a), since Paolo prefers to go by train. At the fifth step, instead, the selected fragment for the payment protocol is the *onLinePayment* exposed by the *Bank Service* DO and depicted in Figure 5(b), to meet the wish of Paolo of paying with his bank account.

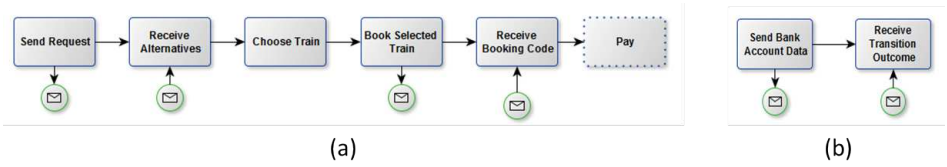

Fig. 5: trainBooking and onLinePayment Fragments.

4 Formal Framework

In this Section we introduce the formal definitions of the elements of our approach and we show how the incremental service composition technique is defined and how it can be automatically resolved by using planning techniques. A system, such as that shown in Section 2, is modeled through a set of entities, each of which is represented by a *domain object* that is formalized as following:

Definition 1. *(Domain Object) A Domain Object is a tuple* $o = \langle \mathcal{CL}, \mathcal{F} \rangle$*, where:*

- $\mathcal{CL}$*, the core layer, represents the behavior of an entity;*
- $\mathcal{F}$ *is a set of services (i.e., process fragments) that an entity exposes and that can be used by other entities.*

In Figure 1, the *FlexiBus Manager* is an example of a DO with its two layers, namely the *Core layer* and the *Fragments layer*. The core layer models all the basic ingredients that make the domain object an independent unit. It essentially represents a set of processes, which configure the behaviors that the entity can execute. As we said, the processes can be not defined in a total and static way. The abstract activities are used to let processes have *opening points* to the outside.

Definition 2. *(Core Layer) The* Core Layer $\mathcal{CL}$ *of a Domain Object* o *is a tuple* $\mathcal{CL} = \langle L, L_0, E_{IN}, E_{OUT}, T \rangle$*, where:*

- L *is the state of a domain object. It is defined through a set of couple* (V, v) *where:*
  - V *is the name of a variable describing a particular aspect of the domain object (e.g., current location of a user, bus delay, availability of a Flexi Bus);*
  - v *is the current value of the variable* V *at a specific execution time.*
  - $L_0 \subseteq L$ *is the initial state;*
- E_{IN} *is a set of events (the "input" events to the object);*
- E_{OUT} *is a set of events (the "output" events from the object), such that* $E_{IN} \cap E_{OUT} = \emptyset$*;*

– $T \subseteq L \times E_{IN} \times L \times 2^{E_{OUT}}$ *is a transition relation.*

We use the same model for both processes and process fragments (later in this section we call both 'fragments'). We remark that processes represent the behaviors of entities. Differently, fragments are pieces of process knowledge provided by entities to advertise their services to other entities around. For example, the *FlexiBus Manager* domain object provides a fragment, namely the *BookFlexiBus*, specifying how to perform the booking of a Flexibus. Now, any other process can perform the Flexibus booking by embedding and executing this fragment, following a specific refinement process. Formally, fragments are state transition systems, where each transition corresponds to a particular fragment activity. In particular, we distinguish four kinds of activities: input and output activities model communications between processes; concrete activities model internal elaborations; and abstract activities correspond to abstract tasks in the process. Abstract activity is what makes process structure dynamic. A process fragment is defined as follows:

Definition 3. *(Process Fragment) Process fragment is a tuple* $p = \langle S, S_0, A, T, Goal \rangle$*, where:*

– S *is a set of states and* $S_0 \subseteq S$ *is a set of initial states;*
– $A = A_{in} \cup A_{out} \cup A_{con} \cup A_{abs}$ *is a set of activities, where* A_{in} *is a set of input activities,* A_{out} *is a set of output activities,* A_{con} *is a set of concrete activities, and* A_{abs} *is a set of abstract activities.* A_{in}*,* A_{out}*,* A_{con}*, and* A_{abs} *are disjoint sets;*
– $T \subseteq S \times A \times S$ *is a transition relation;*
– $Goal : A_{abs} \rightarrow 2^L$ *is the goal labeling function.*

On the basis of the previous definitions, a service-based application simply consists in a directed acyclic graph of DOs connected in a proper way, i.e., where the events produced by a DO are fully captured by all DOs monitoring it. This graph constructs a conceptual network of DO organized in a hierarchic way, as shown in Figure 2.

Definition 4. *(Service-based Application) A Service-based Application (SBA) is a pair* $\Delta = \langle O, H \rangle$*, where:*

– O *is a set of domain objects representing the entities of the system;*
– $H \subseteq O \times O$ *is a hierarchical direct relationship such that:*
  (a) it must be acyclic, i.e., there must exist no sequence $o_1, o_2, \ldots, o_n$ *of domain objects in* O *such that* $o_1 = o_n$ *and* $\forall i : \langle o_i, o_{i+1} \rangle \in H$*, and*
  (b) $\forall \langle o_1, o_2 \rangle \in H : E_{OUT}(o_1) \subseteq E_{IN}(o_2)$.

In the previous definition we have formalized how domain objects can be organized to configure a SBA. In the following, we formalize the execution of a single domain object, the execution of the entire SBA and, the activity refinement process. The execution of a domain object consists in the execution of its fragments, when they are invoked and/or in the execution of its internal processes, when they are triggered. In both cases, executing a fragment/process means to execute the sequence of activities from which they are composed.

Definition 5. *(Domain Object Configuration) We define a domain object configuration as a non-empty list of tuples* $E_o = (p_1, a_1), (p_2, a_2) \ldots (p_n, a_n)$*, where:*

– p_i *are process fragments or internal processes;*
– $a_i \in A(p_i)$ *are activities in the corresponding process fragments, with* $a_i \in A_{abs}(p_i)$ *for* $i \geq 2$ *(i.e., all activities that are refined are abstract);*

Examples of tuples are those depicted in the levels of Figure 4. If, during the execution, an abstract activity is meet, this has to be refined by replacing it with a composition of other fragments. The refinement will modify the state of the domain object stating that the goal of the abstract activity has been reached. The advantage of performing the composition at run-time is twofold: (i) available fragments are not always known at design time (e.g., a new parking payment procedure may be activated and the corresponding fragment added to the system), and the composition strongly depends on the current state of an entity (e.g., the end-user change his preferences or some features in his profile). Composed fragments may also contain abstract activities which requires further refinements during the process execution. The execution of the whole service-based application Δ is defined by the current state of each domain objects, by the domain objects executions in the system, and by the set of available fragments provided by the different domain objects.

Definition 6. *(Service-based Application Configuration) Given a set O of domain objects, we define a service-based application configuration for O as a triple $\mathcal{S} = \langle \mathcal{I}, \Gamma, \mathcal{F} \rangle$, where:*

- $\mathcal{I} \in L(o_1) \times \ldots \times L(o_n), o_i \in O$ *represents the set of current states of the domain objects;*
- $\Gamma \in E_{p_1} \times \ldots \times E_{p_n}$ *represents the configurations of the domain objects;*
- $\mathcal{F}$ *is the set of available fragments provided by all domain objects.*

For lack of space, we do not give a formal definition of the evolution of a SBA configuration. Intuitively, a SBA evolves in three different ways. First, through the execution of the behaviors of domain objects: this happens according to the standard rules of business process execution. Second, through the entrance (and exit) of new entities into the system: each new entity cause the introduction of a new domain object in Γ and the instantiation of the corresponding relations; moreover, since entities can bring new fragments, it also corresponds to the extension of the set $\mathcal{F}$. Third, through the refinement of abstract activities, which will be discussed in detail in the following.

Definition 7. *(Abstract Activity Refinement) An abstract activity refinement is a tuple $\xi = \langle \mathcal{S}, \mathcal{G} \rangle$, where $\mathcal{S}$ is the current SBA configuration and $\mathcal{G}$ is a goal over the state of the domain object o to which the abstract activity belongs.*

An abstract activity refinement ξ is a process R that is obtained as the composition of a set of fragments in $\mathcal{F}(\mathcal{S})$. When executed from the current domain object configuration E_o, R ensures that the resulting domain object configuration satisfies the goal $\mathcal{G}(\xi)$. This means that the composition is obtained by chaining the new fragment in the configuration of the domain object o: $E'_o = (R, a_0) \cdot E_o$. For example, looking at the steps 3 and 4 of the Figure 4 we can see the current configuration of the FBM, representing E_o, at the step 3. The *Pay* abstract activity is a_0, the input for the refinement process. When executed, the refinement process R leds to place the fragment shown at the step 4 in substitution of the abstract activity, bringing to a new configuration, exactly E'_o.

5 Implementation

In this Section we discuss the implementation of a *prototype* corresponding to the SBA described in Section 3 using our approach. The implementation language that we have used is Java. The aim of the prototype is to show an object-oriented representation of domain objects and their relations. The incremental service composition, instead, is dynamically realized by exploiting existing dynamic composition techniques such as those exposed in [4]. We primarily focused on the main ingredients of the model presented in this paper. The integration with the adaptation engine and the embedding of fragment compositions in replacing abstract activities are out of the scope of this paper and they are considered as future work. In the following paragraph we discuss the core layer implementation of a DO and an example of a fragment implementation.

Core Layer Implementation. In this Section we take the FBM code (see Figure 1) as an example of a core layer implementation. We present some chunk of java code implementing the three main parts of the FBM core layer introduced in Section 3, namely the structure, the interface and the behavior. The structure is made of variables and relations, such as those shown in Figure 6(a). The FBM maintains in its state information about the *routes*, the *flexiBuses* data and the *ticketStatus* for each seat of each flexi-bus (lines from 14 to 18). The relation *fbInstances* in lines 21-22, instead, is used by the FBM to manage the flexi-bus instances representing the real flexi-buses around in the city. The Figure 1 shows that the FBM has three main processes, namely the *paymentRequestManagement*, the *routePlanner* and the *flexiBusBooking*. They can be implemented as class methods, as depicted in Figure 6(b). Finally, in Figure 6(c) and 6(d), the interface implementation of the core layer is exposed. The Figure 6(c) reports the ports on which the domain object publishes its events or forwards some notification. They are identified by the *@Port* annotation and by the empty implementation. An example of a publication of an event on a port is reported in Figure 6(d), at line

Fig. 6: Core Layer implementation

83, where the *flexiBusServiceUnavailable* port is called. The Figure 6(d) displays both a *Service Notification* example, from line 78 to line 84, and a *Subscription Definition* example, from line 86 to line 90. The service notification is represented by the *@ServiceNotification* annotation and it is triggered by a notification from the service specified by the annotation parameters. The service subscription, instead, consists in a method labeled with the *@Subscribe* annotation whose parameters *event* and *type* say which is the monitored event and to which domain object it belongs (lines 86-87).

Fragments Layer Implementation. The fragments layer models the set of services that the domain object externally exposes. A fragment can be seen as a *class* whose methods implement the activities making the fragment. In Figure 7 we shown a basic implementation of the fragment *TicketPaymentRequest* of ther FBM. It is important to notice the following aspects.

```
3  @Fragment
4  public class TicketPaymentRequest implements FragmentsSet{
5
6      private FlexiBusManagerCore manager;
7      private Response response;
8      private Status ticketStatus = Status.NOT_PAID;
9
10     public TicketPaymentRequest(FlexiBusManagerCore manager) {
11         super();
12         this.manager = manager;
13     }
14
15     public void execute(Data data) {
16         this.sendBookingCodeAndUserData(data);
17         if (this.response == Response.ACK) {
18             this.pay();
19         }
20     }
21
22     private void sendBookingCodeAndUserData(Data data) {
23         this.manager.paymentRequestManagement(this, data);
24     }
25
26     protected void receiveReply(Response response){
27         this.response = response;
28     }
29
30     @abstractActivity(Goal = "manager.ticketStatus = paid")
31     private void pay(){}
32
33 }
```

Fig. 7: Fragment implementation.

The fragment holds a reference to the core layer of the domain object to which it belongs, namely the *FlexiBusManagerCore* reference, at line 6, that is used to interact with the core layer processes. Lines from 15 to 20 implement the method *execute*, which is used in a service composition containing the current fragment to start its execution. Lines 30 and 31, instead, show how the abstract activity *pay* is defined. An abstract activity is essentially a method with no implementation. It is annotated with a specific annotation, *@abstractActivity* (line 30), stating that the method maps an abstract activity defined by the *goal* parameter of the annotation. In line 18, the abstract activity *pay* is called inside the 'execute' method. Being abstract it will not be executed but the adaptation engine is called and notified that an abstract activity needs to be refined, on the

basis of its goal. At this point the engine will provide the fragments composition for replacing the abstract activity.

6 Related Work

In last years, different approaches have been proposed to provide relevant techniques for the modeling of services in a suitable way for making efficient dynamic service composition. From the scenario discussed above, the need of being able to define services in a way that they can dynamically adapt to the context, when this is discovered or when it changes, is emerged.

An approach is presented by Hull et al. [11] in their work about *Business Artifacts*. It consists in the definition of a formal/theoretical framework for defining conceptual entities, the artifacts, related to the execution of services whose operations influence the entities evolution, as they move through the business's operations. However, this approach deal not with dynamic service composition needs but it focuses only on service modeling aspects. Yu et al. propose MoDAR [24], an approach on how to design dynamically adaptive service-based systems. Essentially they propose a method to simplify business processes by using rules to abstract their complex structure and to capture their variable behavior. However, in dynamic context, to rethink rules every time is to expensive to manage continuous and unpredictable changes. In [12], the authors tackled the problem of the unpredictable execution of service-based applications. In particular, they focused on how to evolve a running service composition. To deal with this purpose they propose a way for modeling artifacts corresponding to composite services that can change at runtime. However, the software engineer intervention is needed to manipulate the runtime model of services. Moreover, the adaptation and application logics are mixed making the model not so flexible. In [7] the authors present DAMASCo, a framework managing the discover, composition, adaptation and monitor of services in a context-aware fashion by taking into account semantic information rather than only the syntactic one. Since they address the problem of making the reuse of software entities more agile and easy to model, they focus especially on the adaptation of pre-existing software entities that are used during the developing of SBA. Also the approach presented in [16] focuses on the need of explicitly manage the context in the composition of web services, to address the problem of semantic heterogeneities between services. The authors present a context-based mediation approach that allows services both to share a common meaning of exchanged data and to automatically resolve conflicts related to the semantic of the data, by using context-based annotations which offer an optimized handling of the data flow. It would be interesting to use the approaches [7] and [16] in the management of the composition of fragments coming from the different DOs and the definition of the data flow between them. In [10] the concepts of goals and plans are introduced in the business processes modeling with the purpose of extending the standard BPMN to make the BPM more flexible and responsive to change. However, even if plans are selected and executed at runtime, they are defined at design time together with the relations with the goals they can satisfy. [9] is a framework for, among other things, the management of the integration of services in the business processes implementation's process to speed the implementation and deployment phases. Services' integration is realized in a plug-and-play manner in which activities are selected from a repository and then dropped into a process. However, as regards the runtime adaptation of processes, in this approach only ad-hoc modifications are managed.

The approaches related to the automation of service composition processes can essentially be grouped into two main categories, *service orchestration* and *service choreography*. The most important in the first category is BPEL [6], which besides composing services, expresses relationships between multiple invocation and it employs a distributed concurrent computation model. WS-CDL [17] is targeted at composing interoperable, long-running, peer-to-peer collaborations between service participants with different roles, as defined by a choreography description. Nonetheless, these approaches refer essentially only to the syntactic aspects of services modeling, thus they provide a set of rigid services that cannot adapt to a changing environment without the human intervention. The vision underlying WSs [15] is to describe the various aspects of services by using explicit, machine-understandable semantics, and, as such, to automate all the stages of the WS lifecycle. OWL-S [14] is an attempt to define

an ontology for the semantic markup of services intended to enable the automation of WS discovery, invocation, composition, interoperation and execution monitoring by providing appropriate semantic descriptions of WSs. The WS Modeling Ontology (WSMO) [23] is an attempt to create an ontology of aspects related to semantic WSs aiming to solve the integration problem. WSCI [2], literally *Web Services Choreography Interface*, is devoted to the description of the messages exchanged between the web services in a choreography and external services. This means that WSCI mainly focuses on the observable behavior of services without deal their internal process. The *Business Process Management Language* (BPML) [1] is also a standard within the service choreography category. To the contrary to WSCI, BPML is a language for the modeling of the internal processes of services in a way which can be supported and executed by business process management systems. It is possible a combined use of WSCI and BPML to design the interface and the business process of a service, respectively. The *Unified Service Description Language* (USDL) [22] has been developed to offer a language for the description of services independently from their belonging domain. The authors' purpose is to allow the definition of all kinds of services from the *Internet of Services* (IoS) perspective.

Discussion. We conclude this section with a discussion in which we try to point out the advantages of the proposed approach. As regards the standard approaches of service composition, such as those of orchestration and choreography, they have have some crucial limitations. A major problem of these approaches is that most of them are based on the assumption that during the composition requirements specification, the application designer knows the services to be composed. Besides, some of them, such as [17], [6], remain focused on the syntax level without consider the semantic aspects of composition, which are, instead, necessary in context-based applications. Others approaches like [15], [14], [23], [2] and [1], have introduced the management of semantic knowledge in their models to drive the services' composition and interoperation but, despite this, they do not allow processes to be defined at runtime, through dynamic service composition. [7], [16] and [10] allow for very efficient management of data and control flows, with their methodologies for the sharing of the common domain and context of services which have to be composed, by overcoming implicit discrepancies existing between heterogeneous services. However, the adaptation's strategies applied in these approaches are defined at design-time.

The DO approach, instead, offers a lightweight-model, with respect to the existing languages for service composition. It can be implemented with every object-oriented languages and it is more flexible and able to define both orchestrations and choreographies thanks to its hierarchical organization of DOs. Moreover, the model explicitly handle the context by managing the dynamicity of services, which can enter or leave the system in any moment, with a flexible connection strategy between DOs that exploit the *publish-subscribe paradigm*. Unlike specifications of traditional systems, where the behaviors are static descriptions of the expected run-time operation, our approach allows the application to define dynamic behaviors. This is made through the use of *abstract activities* representing opening points in the definition of processes, which allow the services to be refined when the context is known or discovered. The *bottom-up approach* for the activities' refinement allows *fragments*, once they are selected for the composition, to climb the DO's hierarchy to be embedded in the running process. Besides, the adaptation strategies are defined at runtime, so that exactly the available services are considered for the composition.

7 Conclusion

In this paper we have introduced an approach to enable a dynamic and incremental service composition, in highly dynamic environments. As future work, we plan to extend the model with the other techniques for the adaptation of SBAs exposed in [4] and to integrate it with our adaptation engine of [21]. We also plan to implement a process for the embedding of fragment compositions in replacing abstract activities in a dynamic and optimized manner. As concern the prototype, our intention is to implement a complete final version and to perform evaluations to assess the approach applicability. Finally, we are also reasoning on the possibility of realizing a design tool for the automation of SBAs design and implementation using DO.

Acknowledgment

This work is partially funded by the 7th Framework EU-FET project 600792 ALLOW Ensembles.

References

[1] Bpml.org. business process modeling language (bpml). *Available at http://www.bpmi.org*, 2002.
[2] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, and et al. Web service choreography interface (wsci). *Available at http://www.w3.org/TR/wsci*, 2002.
[3] P. Bartalos and M. Bieliková. Automatic dynamic web service composition: A survey and problem formalization. *Computing and Informatics*, 30(4):793–827, 2011.
[4] A. Bucchiarone, A. Marconi, M. Pistore, and H. Raik. Dynamic adaptation of fragment-based and context-aware business processes. In *ICWS*, pages 33–41, 2012.
[5] A. Bucchiarone, A. Marconi, M. Pistore, P. Traverso, P. Bertoli, and R. Kazhamiakin. Domain objects for continuous context-aware adaptation of service-based systems. In *ICWS*, pages 571–578, 2013.
[6] OASIS WSBPEL Tecnical Committee. Web services business process execution language, version 2.0. *Available at http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0, 2007*, 2007.
[7] J. Cubo and E. Pimentel. Damasco: A framework for the automatic composition of component-based and service-oriented architectures. In *ECSA*, pages 388–404, 2011.
[8] H. Eberle, T. Unger, and F. Leymann. Process fragments. In *OTM Conferences (1)*, pages 398–405, 2009.
[9] K. Göser, M. Jurisch, H. Acker, U. Kreher, M. Lauer, S. Rinderle, M. Reichert, and P. Dadam. Next-generation process management with adept2. In *BPM (Demos)*, 2007.
[10] D. A. P. Greenwood. Goal-oriented autonomic business process modeling and execution: Engineering change management demonstration. In *BPM*, pages 390–393, 2008.
[11] R. Hull, E. Damaggio, R. De Masellis, F. Fournier, M. Gupta, F. Terry Heath, S. Hobson, M. H. Linehan, S. Maradugu, A. Nigam, P. Noi Sukaviriya, and R. Vaculín. Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In *DEBS*, pages 51–62, 2011.
[12] M. Hussein, J. Han, Y. Yu, and A.Colman. Enabling runtime evolution of context-aware adaptive services. *IEEE International Conference on Services Computing*, 2013.
[13] Simon Laws, Mark Combellack, Raymond Feng, Haleh Mahbod, and Simon Nash. *Tuscany SCA in Action*. Manning Publications, 2011.
[14] D. L. McGuinness and F. van Harmelen. Owl web ontology language overview [online]. *Available at http://www.w3.org/TR/owl-features/.*, 2004.
[15] Sheila A. McIlraith, Tran . Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.
[16] M. Mrissa, C. Ghedira, D. Benslimane, Z. Maamar, F. Rosenberg, and S. Dustdar. A context-based mediation approach to compose semantic web services. *ACM Trans. Internet Techn.*, 8(1), 2007.
[17] D. Burdett N. Kavantzas and G. Ritzinger. Wscdl v1.0. *Available at http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/*, 2004.
[18] OpenSOA. Service component architecture specifications. http://www.oasis-opencsa.org, 2007.
[19] C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, 2003.
[20] M. Pistore, P. Traverso, M. Paolucci, and M. Wagner. From software services to a future internet of services. In *Future Internet Assembly*, pages 183–192, 2009.
[21] H. Raik, A. Bucchiarone, N. Khurshid, A. Marconi, and M. Pistore. Astro-captevo: Dynamic context-aware adaptation for service-based systems. In *SERVICES*, pages 385–392, 2012.
[22] S. Srividya, A. Bansal, L. Simon, and T.Hite. Usdl: A service-semantics description language for automatic service discovery and composition. 2009.
[23] WSMO. Wsmo working group. *Available at http://www.wsmo.org.*
[24] J. Yu, Q. Z. Sheng, and J. K. Y. Swee. Model-driven development of adaptive service-based systems with aspects and rules. In *WISE*, volume 6488 of *Lecture Notes in Computer Science*, pages 548–563. Springer, 2010.