Journal of Internet Services and Applications

RESEARCH **Open Access**

Dynamic adaptation of service-based applications: a design for adaptation approach

Martina De Sanctis[1*], Antonio Bucchiarone[2] and Annapaola Marconi[2]

Abstract

A key challenge posed by the Next Generation Internet landscape is that modern service-based applications need to cope with *open* and *continuously evolving* environments and to operate under dynamic circumstances (e.g., changes in the users requirements, changes in the availability of resources). Indeed, dynamically discover, select and compose the appropriate services in such environment is a challenging task. Self-adaptation approaches represent effective instruments to tackle this issue, because they allow applications to adapt their behaviours based on their execution environment. Unfortunately, although existing approaches support run-time adaptation, they tend to foresee the adaptation requirements and related solutions at design-time, while working under a "closed-world" assumption. In this article our objective is that of providing a new way of approaching the *design*, *operation* and *run-time adaptation* of service-based applications, by considering the adaptivity as an intrinsic characteristic of applications and from the earliest stages of their development. We propose a *novel design for adaptation approach* implementing a complete lifecycle for the continuous development and deployment of service-based applications, by facilitating (i) the continuous integration of new services that can easily join the application, and (ii) the operation of applications under dynamic circumstances, to face the *openness* and *dynamicity* of the environment. The proposed approach has been implemented and evaluated in a real-world case study in the mobility domain. Experimental results demonstrate the effectiveness of our approach and its practical applicability.

Keywords: Service-based adaptive applications, Next generation internet, Design for adaptation, Incremental service composition, Smart mobility

1 Introduction

The Internet of Services (IoS) is widespread and it is becoming more and more pervasive, due to the trend of delivering *everything as a service* [1], from applications to infrastructures, passing through platforms [2]. Furthermore, the IoS is envisioned as one of the founding pillars of the Next Generation Internet [3], together with new metaphors, such as those of the Internet of Things (IoT) and the Internet of People (IoP) [4].

*Correspondence: martina.desanctis@gssi.it
[1]Gran Sasso Science Institute, Computer Science department, Viale Francesco Crispi, 67100 L'Aquila, Italy
Full list of author information is available at the end of the article

In last decades, the aim of service-oriented computing has been that of encouraging the creation and delivery of services. Automated service composition is a powerful technique allowing to compose and reuse the existing services as building blocks for new services (and applications) with higher-level functionalities.

To date, service-based applications are employed in a multitude of domains, such as e-Health, smart homes, e-learning, education, smart mobility and many others. Additionally, the role played by companies and organizations is also considerable. They are publicly providing their services to allow third-party developers to exploit them in defining new services, thus enhancing their accessibility [5] (e.g., Google Maps, Paypal). This

Springer Open

© The Author(s). 2020 **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

is of relevant importance in the Future Internet scenario, since it implies the availability of a multitude of reliable services offering even complex functionalities. Different organizations are building on this trend to provide online platforms for the management of well-defined RESTful APIs—REpresentational State Transfer Application Program Interface, through which these services can be accessed. For instance, ProgrammableWeb [1] has now more than 10,000 APIs in its directory. As a consequence, both researchers and practitioners are highly motivated in defining solutions allowing the development of service-based applications, by exploiting existing available services.

In this scenario, service-based applications must face the increased *flexibility* and *dynamism* offered by modern service-based environments. The number and the quality of available services is continuously increasing and improving. This makes service-based environments *open* and *highly dynamic*, since service–oriented computing takes place in an "open world" [6].

These premises demand *self-adaptive* service-based applications, that is, applications able to both *adapt* to their context (i.e., the currently available services) and *react* when facing new contextual situations (e.g., missing services, changes in the user requirements and needs). However, *there are still major obstacles that hinder the development and potential realization of service computing in the real world* [5]. In fact, the latest Next Generation Internet vision further challenge the IoS paradigm. Service-oriented computing has to face the ultra large scale and heterogeneity of the Future Internet, which are orders of magnitude higher than those of today's service-oriented systems [7].

In this context, *self-adaptation* is still one of the main concerns. Many service-based methodologies and approaches have been proposed with the aim of increasing the flexibility of applications and supporting their *adaptation* needs. They span from *microservices* [8, 9], to *DevOps* (e.g., [10]), passing through *dynamic software product lines* [11], to name a few. Nevertheless, none of them is specifically meant for open environments, where the available services might not be known a priori and/or not available at execution time. However, to perform accurately, service-based applications must be aware of the specific execution environment during their execution, thus operating differently for different contextual situations. The openness of the environment makes traditional adaptation mechanisms no longer sufficient. Differently from applications where traditional adaptation mechanisms can be used, the IoS requires applications that are *adaptive by design*. These premises motivated the work presented in this article about a novel *design for adaptation approach of service-based applications*. To this aim, the adaptation must be hold by a *coherent design approach*, supporting both the *definition* and the *application* of adaptation.

In very general terms, the idea of the approach consists in defining the complete lifecycle for the *continuous development* and *deployment* of service-based applications, by facilitating (1) the *continuous integration of new services* that can easily join the applications, and (2) the applications operation under *dynamic circumstances*, to face the *openness* and *dynamicity* of the environment.

This article is an extension of [12, 13] where we have introduced and formalized a design for adaptation approach of service-based applications relying on incremental service composition. The novel contributions of this article are: (i) the *overall lifecycle of the design for adaptation approach* that gives a complete overview of the different perspectives (i.e., modeling, adaptation, interaction) of the approach, the involved components (e.g., artefacts, performed activities, engines) and the *connections among them*, while also considering the role played by the potentially involved actors; (ii) presenting the approach as a whole, gave us the possibility to shape a clear *positioning* of the presented approach in the literature about existing approaches for the design of service-based applications and their dynamic adaptation; (iii) further details about previously unpublished constructs of the approach, and *extended experimental results* that include *new elements on the approach efficiency*.

The article is organized as follows: a motivating scenario and research challenges are described in Section 2. In Section 3 a high level overview of the whole approach in introduced. The subsequent two sections present the novel design for adaptation approach, in Section 4, and how the defined applications operate at run-time, in Section 5. Validation results are reported in Section 6 where the approach is applied to a real case study, in the Smart Mobility domain. Section 7 describes the overall lifecycle of the design for adaptation approach. Related work are discussed in Section 8. Section 9 discusses the open issues raised by the approach, while Section 10 concludes the article with final considerations and directions for future work.

2 Motivating scenario and research challenges

In this section we introduce the travel assistant scenario, in Section 2.1, and the research challenges arising from these applications, in Section 2.2.

2.1 Travel assistant scenario

The travel assistant scenario belongs to the mobility domain, which is particularly suitable to show the challenges of open and dynamic environments. It concerns with the management and operation of mobility

[1] http://www.programmableweb.com

services, within a smart city as well as among different cities/countries. Nowadays, users dispose of a large offer of mobility services that may differ depending on diverse aspects, such as the offered functionalities, the provider, the geographical applicability scope, etc. In addition, mobility services span from *journey planners* to *specific mobility services*, such as those referring to specific transport modes (e.g., bus, train, bikes) or provided by specific transport companies. Moreover, an emerging trend is that of *shared mobility services* that are based on the shared use of vehicles, bicycles, or other means. Mobility services can offer disparate functionalities (e.g., journey planning, booking, online ticket payment, seat reservation, check-in and check-out, user profiling, and so on). Some functionalities may be peculiar to specific services and/or require particular devices (i.e., unlocking a bike from a rack is peculiar for bike-sharing services, and a smart-card might be needed to do it). These services are made available through a large variety of technologies (e.g., web pages, mobile applications), with different constraints on their availability (e.g., free vs. pay).

A *journey organization*, from a user perspective, consists of a set of different mandatory and/or optional phases that must be carried out (e.g., planning, booking, check-in, check-out). While these phases define *what* should be done, *how* they can be accomplished strongly depends on the users requirements and preferences, and from the procedures that need to be followed, as provided by the available mobility services.

Example 1 *A user plans her journey by looking for the available (multi-modal) alternatives satisfying her needs. The journey can be both local, in the context of a city, or global involving different cities/countries. A multi-modal solution can involve different transportation means, each requiring for different procedures to be followed. During the execution, if extraordinary events affect the journey, it can be re-planned and recovery solutions can be suggested to the user. Thus, users need support or the whole travel duration.*

To this aim, different mobility services need to synergistically cooperate. While the idea of an intelligent travel assistant has already been figured out in the past, as for instance in [14], our opinion is that we are still far from making it happens.

2.2 Research challenges

Modern service-based applications need to satisfy different *requirements* to deal with the features of modern execution environments, thus arising the following research challenges:

Applicability in open environments. Applications must be capable to operate in open environments with continuously entering and leaving services. Nonetheless, traditional approaches work under a "closed–world" assumption, although the today scenario is that service–oriented computing happens in an "open world" [6].

Autonomy and heterogeneity of services. Applications must take into account the autonomous nature of the services involved as well as the heterogeneity among services.

Context-awareness. The application must take into account the state of the environment in which it operates, to behave according to it.

Services interoperability. Applications must be capable to propose complex solutions taking advantages of the variety of services. Moreover, different solutions can be applied for the same goal (e.g., user goal), depending on, i.e., the available services or the user requirements. This means that the composition of services must be performed dynamically.

Adaptivity and scalability. The application must be able to react and adapt to changes in the environment that might occur and affect its operations. Moreover, due to the dynamicity of the environment, the adaptation must be postponed as much as possible to the runtime execution of applications, when the environment is known.

User centricity and personalization. Applications must take into account the nature of users, which are proactively involved in the applications they use and increasingly demanding. Applications must provide users with personalized solutions.

Portability. Modern applications should be deployable in different environments without an ad-hoc reconfiguration from the developers.

3 Overview of the approach

The work presented in this article has been inspired from the work presented in [15] where the authors argue that mechanisms enabling adaptation should be introduced in the lifecycle of applications, both in the *design* and in the *run-time* phases. In other words, applications must be *adaptive by design*. They should rely on a *dynamic set of autonomous and heterogeneous services* that are composed dynamically without any a-priori knowledge between the applications and the exploited services. To this aim, three conditions are required, as depicted in Fig. 1:

1 the models adopted for the applications design must allow the definition of dynamically customizable applications behaviors, through the adoption of adequate constructs. This is done in the Modeling phase of Fig. 1, where specific models are used to wrap-up in a uniform way existing or new services in a given domain (Real services wrapping & Value Added Services (VAS) modeling activity).

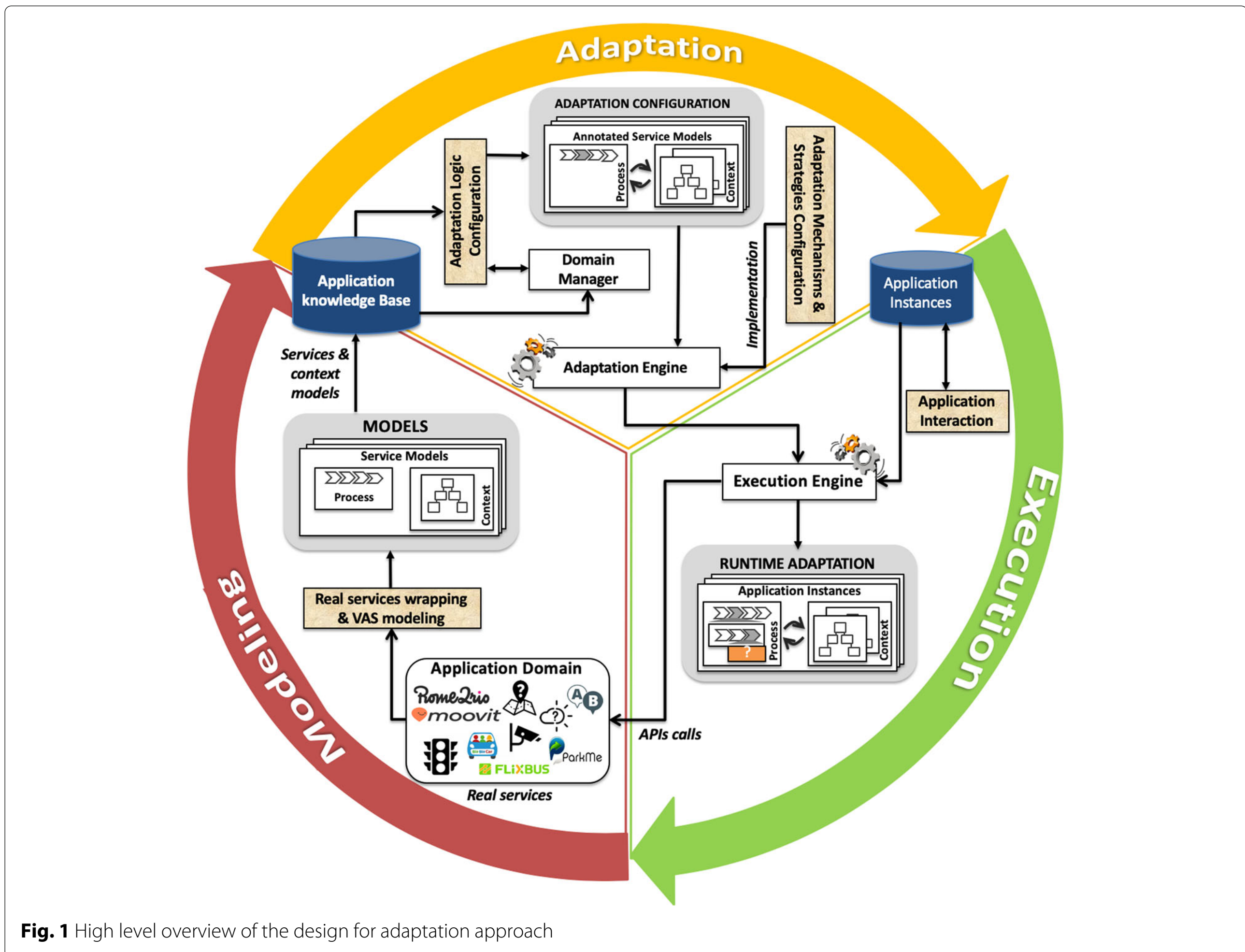

Fig. 1 High level overview of the design for adaptation approach

2 The approach must implement or exploit adaptation mechanisms and strategies whose application allows for a context-aware and dynamic adaptation, during their execution. To this aim, during the Adaptation phase of Fig. 1 adaptation strategies must be implemented (Adaptation mechanisms & strategies configuration activity), while the adaptation logic of the defined applications must be configured accordingly (Adaptation logic configuration activity).
3 In open world, the adaptation must be postponed as much as possible to the Execution phase of applications (Application interaction activity), when the environment is known, without any a-priori definition of adaptive solutions.

Eventually, we specify that the approach is *domain-independent* and it can be applied in multiple domains (e.g., logistic, traveling, entertainment, smart environments). Notably, in [16] its has been applied in the IoT domain. Nonetheless, in this article we only focus on a scenario belonging to the IoS domain.

In the following sections we will deeper illustrate the models of the approach, in Section 4, and the adaptation mechanisms and strategies in Section 5.

4 Adaptive service-based applications: modeling

In our approach is central the use of two separate models, namely the *domain model* and the *domain objects model*, which implement the separation of concerns principle (*adaptation* vs. *application* logic). Keeping the two models separate allows the operational semantic of services (i.e., in the domain model), to be detached from the different implementations that might be provided by a plethora of different concrete services (i.e., in the domain objects model). We start with an overview on the general framework and its models, in Section 4.1. Afterwards we give formal definitions of the models elements, in Section 4.2.

4.1 The design for adaptation approach

In this section, we describe the models, by also mapping each element with a corresponding example within the travel assistant application.

The travel assistant is modeled through a set of *domain objects* representing the services provided by the application (e.g. `Travel Assistant`, `Journey Manager`). In particular, existing or newly defined services can be wrapped-up as domain objects. Wrapping a service as a domain object means shape it in terms of the domain objects components, which we are going to introduce in the following. More precisely, the service's implementation already exists and is made available, e.g., through APIs. The wrapping activity consists in modeling the service in a uniform way, namely as a domain object, in which the provided APIs are exploited. As depicted in Fig. 2, each domain object is characterized by a *core process*, implementing its own behavior, and a set of process *fragments*, representing the functionalities it provides.

Fragments [17, 18] are executable processes that can be received and executed by other domain objects to exploit a specific functionality of the provider domain object. Exposed fragments and the core process communicate through the execution of input/output activities. This concerns the fact that fragments act as an interface for the internal behavior of a domain object, thus they need to interact with the core process to eventually accomplish the functionalities they model. Both core processes and fragments are modeled in Adaptive Pervasive Flow Language–APFL [19].

Unlike traditional application specifications, where services' behavior are completely specified pre deployment, our approach allows the partial specification of the expected operation of domain objects. Indeed, APFL handles the use of **abstract activities** labeled with *goals* and acting as open points enabling the customization and adaptation of processes (see the white activities with goals in Fig. 2). These activities are then *refined at run-time* according to the fragments offered by the other domain objects in the application. We illustrate this notion with a simple example.

Example 2 *In Fig. 3 we show a portion of the travel assistant made by a subset of its services and their potential dependencies. The* `Journey Planners Manager` *can partially define the functionality allowing the planning of a journey. Then, different journey planners can join the application and publish different planning procedures, covering areas of varying size and boundaries (i.e., local and global journey planners). Only at run-time, when the user's source and destination points are known, the* `Journey Planners Manager` *will discover those domain objects modeling journey planners, with their fragments, and it will exploit them to refine its abstract activity and to eventually get the list of available multi-modal alternatives for the specified input.*

An important aspect of the design model that strongly supports the application's dynamicity consists in the fact that *abstract activities can be used in the core process of a domain object* as well as in the fragments it provides.

In the first case, the domain object leaves underspecified some activities, in his own behavior, that are automatically refined at run-time. The latter case is more complex, and it enables a *higher level of dynamicity*, since it allows a domain object to expose a *partially* specified fragment whose execution does not rely only on communications with its core process but also on fragments

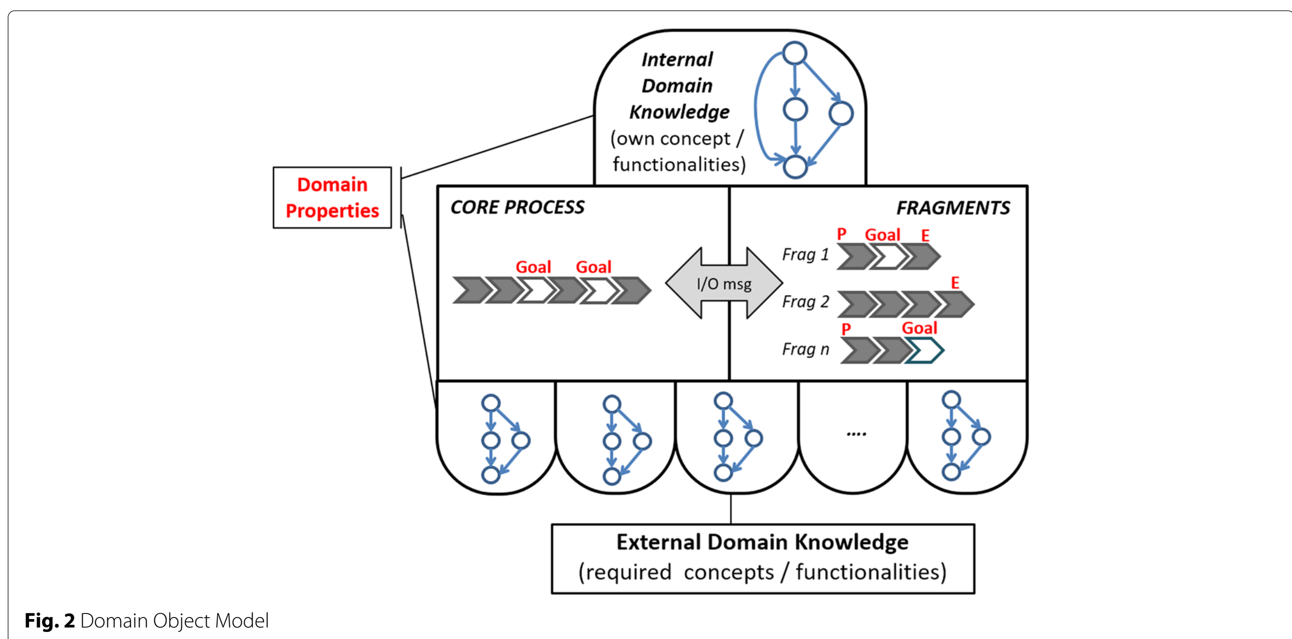

Fig. 2 Domain Object Model

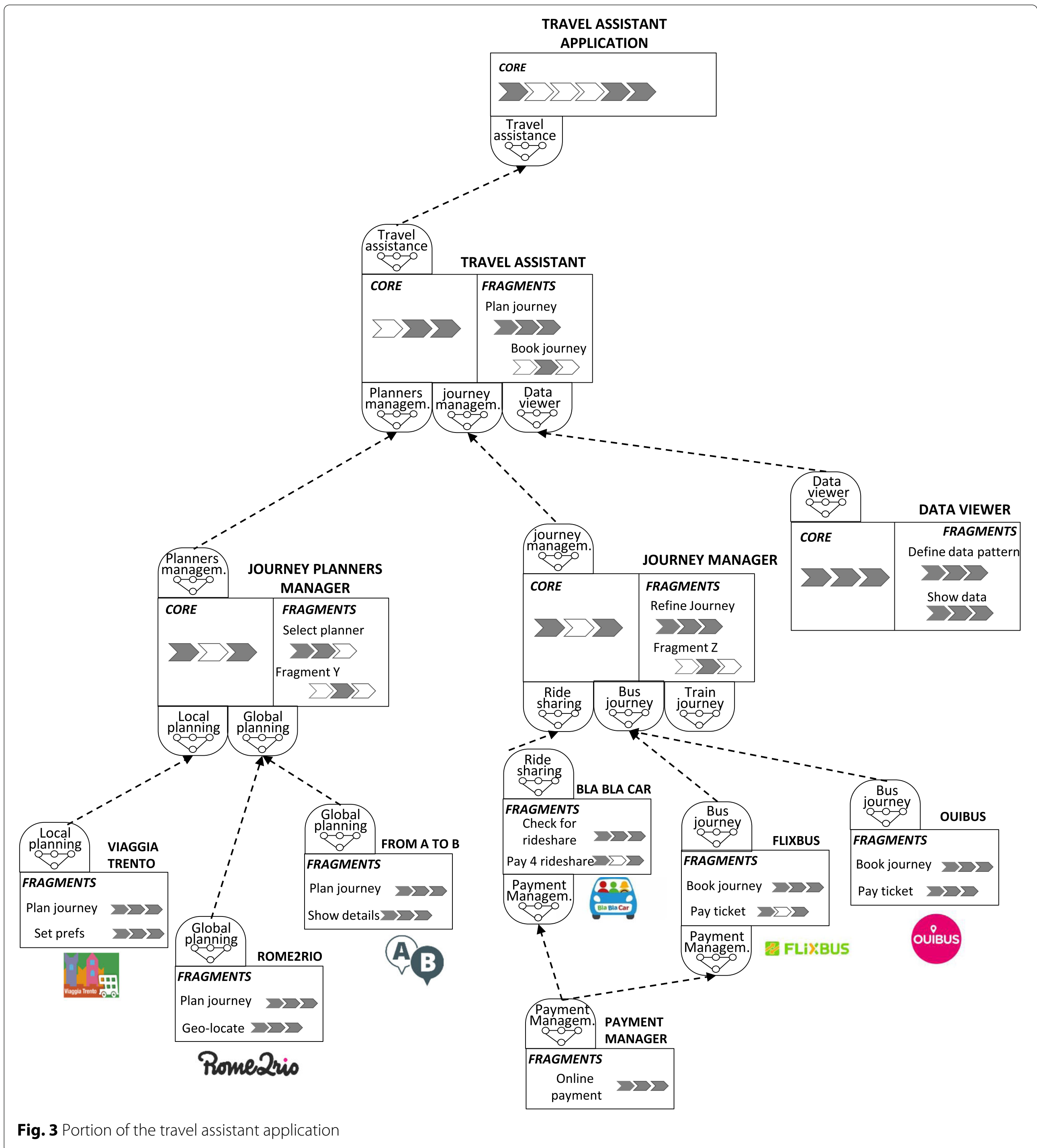

Fig. 3 Portion of the travel assistant application

provided by other domain objects, thus enabling a *chain of refinements*. This will be shown and discussed in Section 5.

These dynamic features rely on a set of *domain concepts* describing the operational environment of the application, on which each domain object has a partial view. In particular (see Fig. 2), the ***internal domain knowledge*** captures the behavior of the domain concept implemented by the domain object, while the ***external domain knowledge*** represents domain concepts that are required to accomplish its behavior but for whose implementation it relies on other domain objects. The domain knowledge (both internal and external) makes the domain model. It is defined by *domain properties*, each giving a high-level

representation of a domain concept (e.g. journey planning, ride-sharing journey). Domain properties are modeled as *State Transition Systems* (STS) evolving as an effect of the execution of service-based applications, or because of exogenous events in the operational context [20, 21]. At this point we must clarify that even if in Fig. 2 we show the domain properties as part of the domain object, which is actually true, we say that domain properties exist independently of the domain objects implementing or relying on them, if any[2]. Indeed, they are identified and defined by domain experts before the application is developed (i.e., before domain objects are designed). Each STS is obtained by analyzing the behavior of those services that will implement it. For instance, the `Ride Sharing` STS in Fig. 4 comes from an analysis and an abstraction of the ride sharing services.

In Fig. 4 we provide some examples of (simplified) domain properties and we give a domain property's evolution example in the following.

Example 3 *The* `Travel Assistance` *domain property models the behavior of a travel assistant. First of all, the journey needs to be planned (*`JOURNEY PLANNED` *state), after that a specific request from the user arrives (*`REQUEST RECEIVED` *state). Then, the user receives the list of possible alternatives (*`ALTERNATIVES SENT` *state) and she chooses the preferred solution among them (*`USER CHOICE RECEIVED` *state). At this point her plan can be further refined by considering the transportation means effectively composing the chosen alternative (*`PLAN REFINED` *state). If required by the involved transportation means, the plan can be also booked (*`PLAN BOOKED` *state), otherwise the user can start her journey (*`JOURNEY EXECUTION` *state) until she reaches her destination (*`ASSISTANCE COMPLETE` *state). During the normal behavior of the application, a domain property may evolve as an effect of the execution of a fragment activity (e.g., if the journey planning activity is successful, the travel assistant moves in the state* `JOURNEY PLANNED`*). Otherwise, if something unexpected occurs, a domain property may also evolve as a result of exogenous changes (e.g., because of roadworks the bus is not passing).*

Eventually, a *domain configuration* is given by a snapshot of the domain at a specific time of the journey, capturing the current status of all its domain properties.

The link between the domain model and the domain objects model is given by ***annotations***. Indeed, APFL gives the possibility to relate the execution of processes with the application domain, through the use of annotations on process activities. Annotations represent domain-related information and they implicitly define a mapping between the execution of processes and fragments and corresponding changes in the status of domain properties. Note that, by properly annotating services (i.e., processes in domain objects) and without changing the domain properties, it is easy to add new services implementations (i.e., new domain objects).

Annotations can be of different types. In particular, each abstract activity is defined in terms of the *goal* it needs to achieve, expressed as domain knowledge states to be reached. Then, the annotated abstract activity is automatically refined at run time, by considering (1) the set of fragments currently provided by other domain objects, (2) the current domain knowledge configuration, and (3) the goal to be reached. In particular, goals are defined over the *external* domain knowledge, since they refer to functionalities which belongs to domain properties implemented by other domain objects. They can be defined as disjunctions of conjunctions over states of domain properties, as we will see further on. To show how annotations are defined, in Fig. 5 we report an example of a fragment modeling the functionality of paying for a rideshare (`Rideshare Payment` fragment), as it might be exposed by a ride-sharing mobility service, such as BlaBlaCar [3]. Moreover, in Fig. 6 we give the (partial) APFL listing for the same fragment.

Example 4 *The activity* `Pay for rideshare` *is an abstract activity, represented with a dotted line, labeled with the goal* `G1` *that is defined over the* `Payment Management` *domain property (see lines* ***25-35*** *in Fig. 6). Indeed, the BlaBlaCar service does not implement the online paying, but it relies on external payment services for the secure payment over internet.*

In addition to goal annotations, activities in processes and fragments are annotated with *preconditions* and *effects*. Preconditions constrain the activity execution to specific domain knowledge configurations.

Example 5 *In Fig. 5, the precondition* `P1` *says that, to execute the fragment* `Rideshare Payment`*, the domain property* `RIDE SHARING` *(see Fig. 4) must be in the state* `PICK-UP POINT DEFINED` *(see lines* ***10-16*** *in Fig. 6). This precondition constrains the execution of the* `Rideshare Payment` *fragment only in those configurations in which the driver and the passenger already defined the pick-up point.*

Effects, instead, model the expected impact of the activity execution on the domain and represent its evolution in terms of domain properties events.

[2]For simplifying the graphical representation of complex applications made by different interconnected domain objects, through this article we draw domain properties as part of domain objects.

[3]https://www.blablacar.it/

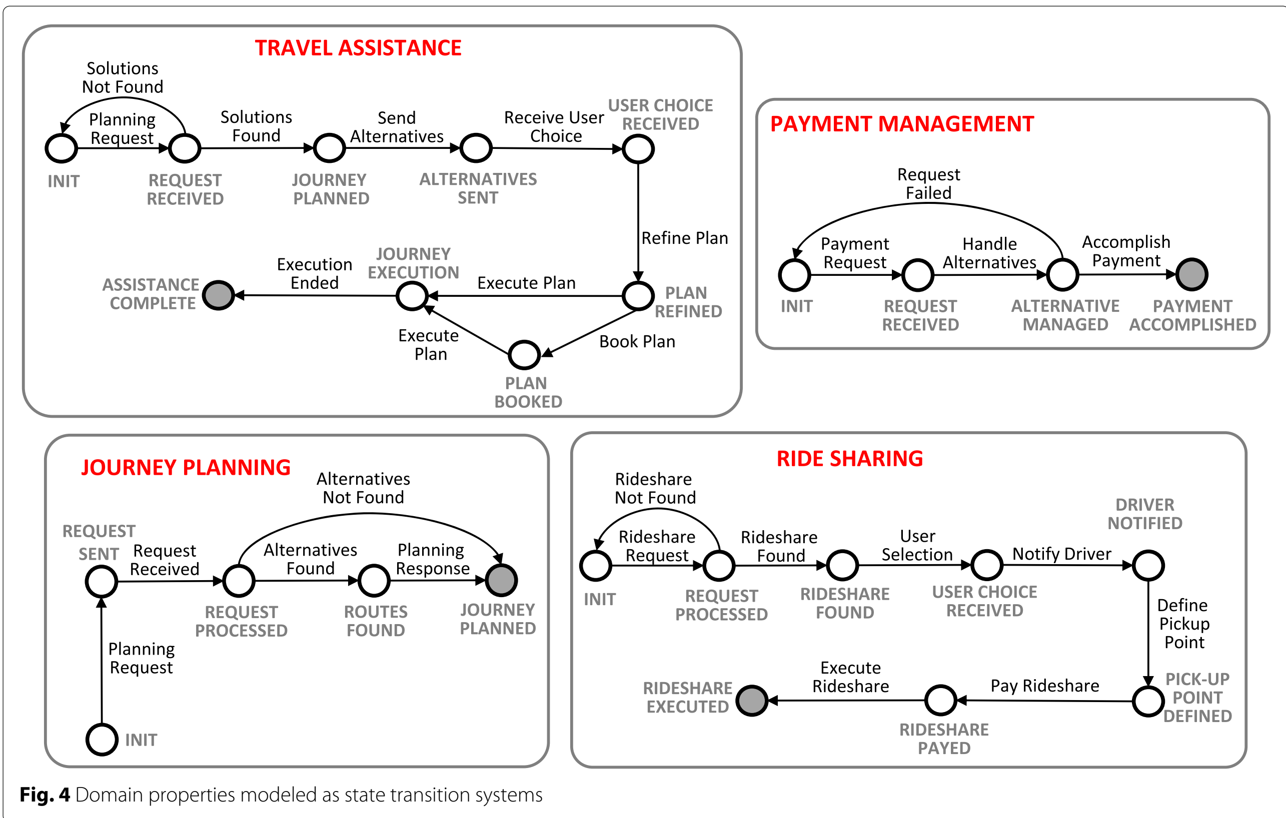

Fig. 4 Domain properties modeled as state transition systems

Example 6 *The effect* `E1` *in Fig. 5, models the evolution of the* `RIDE SHARING` *domain property (see Fig. 4). It is caused by the event* `PayRideshare`*, triggered by the* `Receive payment ack` *activity and it brings the property in the state* `RIDESHARE PAYED` *(see lines* ***38-40*** *in Fig. 6).*

Preconditions and effects are used to model how the execution of fragments is constrained by and evolve the domain knowledge. This information is used to identify the fragment (or composition of fragments) that can be used to refine an abstract activity in a specific domain knowledge configuration.

Example 7 *As shown in Fig. 3, the* `RIDE SHARING` *domain property belongs to the internal domain knowledge of the* `BlaBlaCar` *domain object and to the external domain knowledge of the* `Journey Manager`*. This property can be used to specify goals of abstract activities within the* `Journey Manager` *(e.g. to handle a ride-share journey). Similarly, fragments offered by the* `BlaBlaCar` *domain object are annotated with preconditions and effects on the* `RIDE SHARING` *domain property.*

Potential dependencies (***soft dependencies***, from here on) are established between a domain object and all those domain objects in the application whose modeled domain concept (internal domain knowledge) matches with one of its required behaviors (domain property in its external domain knowledge). Figure 3 shows the soft dependencies (dashed arrows) among some of the domain objects modeling the travel assistant application. A soft dependency between two domain objects becomes a ***strong***

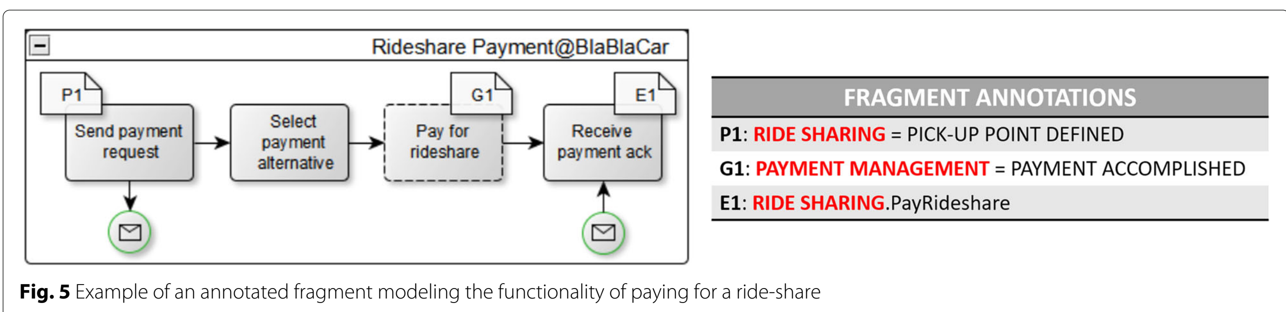

Fig. 5 Example of an annotated fragment modeling the functionality of paying for a ride-share

```
1<?xml version="1.0" encoding="UTF-8"?>
2<tns:fragment id="Rideshare_Payment" xmlns:tns="http://das.fbk.eu/Fragment"
3    xmlns:tns1="http://das.fbk.eu/Annotation"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://das.fbk.eu/Fragment␣Fragment.xsd">
6
7    ...
8
9    <tns:action name="Send_Payment_Request" actionType="input">
10       <tns:precondition>
11           <tns1:point>
12               <tns1:domainProperty dp_name="RideSharing">
13                   <tns1:state>PICK-UP POINT DEFINED</tns1:state>
14               </tns1:domainProperty>
15           </tns1:point>
16       </tns:precondition>
17
18       <!-- Activity's variables here -->
19
20   </tns:action>
21
22   <tns:action name="Select_Payment_Alternative" actionType="concrete">
23   </tns:action>
24
25   <tns:action name="Pay_per_Rideshare" actionType="abstract">
26       <tns:goal>
27           <tns1:point>
28               <tns1:domainProperty dp_name="PaymentManagement">
29                   <tns1:state>
30                       PAYMENT ACCOMPLISHED
31                   </tns1:state>
32               </tns1:domainProperty>
33           </tns1:point>
34       </tns:goal>
35   </tns:action>
36
37   <tns:action name="BBC_PassageResponse" actionType="output">
38       <tns:effect>
39           <tns1:event dpName="RideSharing" eventName="PayRideshare"/>
40       </tns:effect>
41
42       <!-- Activity's variables here -->
43
44   </tns:action>
45
46   <tns:transition>
47       <tns:initialState>ST0</tns:initialState>
48       <tns:finalState>ST1</tns:finalState>
49       <tns:action name="Send_Payment_Request" actionType="input"/>
50   </tns:transition>
51
52   <!-- List of transitions here -->
53
54</tns:fragment>
```

Fig. 6 APFL listing of the Rideshare_Payment fragment

dependency if, during the application execution, they inter-operate by exchanging their fragments and domain knowledge. In Section 5, which is about the execution of service-based applications, we present a run-time scenario and we show how soft dependencies become strong dependencies after the refinement of abstract activities.

Eventually, the resulting adaptive application can be seen as a dynamic network of interconnected domain objects which dynamically inter-operate. In particular, the network is structured as a *hierarchy of domain objects*, where the abstract activities refinement mechanism enables a *bottom-up approach* allowing fragments, once they are selected for the composition, to climb the domain objects' hierarchy to be injected in the running processes. Notice that the external domain knowledge of a domain object is not static since, it can be extended during the execution of domain objects, due to specific operational cases, as we will better see in Section 5. As regards the entrance/exit of new domain objects, the approach explicitly handle the domain by managing the dynamicity of services, which can enter or leave the application at any moment. This is due to the use of the *domain model* that provides an abstract representation of the domain concepts, which can be concretized by different services, each giving their own implementation of a specific concept.

4.2 Models formalization

In this section, we give formal definitions of the core elements of our approach. Firstly we define the *domain model* in Section 4.2.1 and then we formalize the *domain objects model* elements in Section 4.2.2.

4.2.1 Domain model

In this section we formalize the domain model through the definition of the domain property concept as its founding element.

Definition 1 (Domain Property) *A domain property is a state transition system $dp = \langle L, l^0, E, T\rangle$, where: L is a set of states and $l^0 \in L$ is the initial state; E is a set of events; and $T \subseteq L \times E \times L$ is a transition relation.*

We denote with $L(dp)$, $E(dp)$, $T(dp)$ the corresponding elements of dp.

Examples of domain properties are shaped in Fig. 4.

Definition 2 (Domain model) *A domain model is a set of domain properties $C = \{dp_1, dp_2, \ldots, dp_n\}$ with $dp_i = \langle L_i, l^0{}_i, E_i, T_i\rangle$ for every $1 \leq i \leq n$, and such that for every pair $1 \leq i, j \leq n$, if $i \neq j$, then $E_i \cap E_j = \emptyset$.*

The set of all domain states is defined as $L_C = \prod_{i=1}^{n} L_i$ and the initial context state is $l^0{}_C = (l^0{}_1, l^0{}_2, \ldots, l^0{}_n)$. The set of all domain events is $E_C = \bigcup_{i=1}^{n} E_i$. Finally, the transition relation in the domain model is given as T_C such that for every pair of states $(l_1, \ldots, l_n) \in L_C$ and $(l'_1, \ldots, l'_n) \in L_C$, and for every event $e \in E_C$, if $e \in E_i$ then $((l_1, \ldots, l_n), e, (l'_1, \ldots, l'_n)) \in T_C$ iff

$$(l_i, e, l'_i) \in T_i, \text{ and for every} j \neq i \text{ we have} l_j = l'_j.$$

A domain model consists in a set of domain properties. We assume that two distinct domain properties $p_i, p_j \in C$ in a domain model do not intersect. The states of a domain model is the product of its domain properties. A state in a domain model can then be seen as the conjunction of states of domain properties. The events of a domain model is the union of the events of its domain properties. Transitions in a domain model are component-wise: each transition changes the state of at most one domain property.

Given a domain model $C = \{dp_1, dp_2, \ldots, dp_n\}$, it will be convenient to denote with $l_i = \bar{l}\downarrow_{dp_i}$ the projection of state $\bar{l} \in L_C$ onto the domain property dp_i.

4.2.2 Domain objects model

In this section we start by introducing all the elements that form a domain object, then we show how domain objects combine to form an adaptive system.

The domain model previously defined is instrumental in the definitions of internal and external knowledge of domain objects.

A domain object has an internal domain knowledge.

Definition 3 (Internal Domain Knowledge) *An internal domain knowledge is a domain model $\mathbb{DK}_I = \{dp_I\}$ where dp_I is a domain property that represents the domain concept implemented by the domain object.*

For instance, let us consider the FLIXBUS domain object in Fig. 3. Its internal domain knowledge is given by the singleton containing the BUS JOURNEY domain property.

A domain object has also an external domain knowledge.

Definition 4 (External Domain Knowledge) *An external domain knowledge is a domain model $\mathbb{DK}_E = \{dp_1, \ldots, dp_n\}$, where each dp_i, $1 \leq i \leq n$, are domain properties that the domain object uses for its operation but that are not under its own control.*

For instance, in the FLIXBUS domain object in Fig. 3, its external domain knowledge is given by the singleton containing the PAYMENT MANAGEMENT domain property, since the Flixbus service requires for the online booking and payment of the tickets, but it does not implements the payment service. Notice that in general, the external knowledge can contain more than one domain property.

The external domain knowledge and the internal domain knowledge are domain models. Hence, they have a set of states, and set of events, and a transition relation as specified in Definition 2. For convenience, we denote $\mathbb{L}_E$ and $\mathbb{E}_E$ the set of states and the set of events in the external domain knowledge. We also denote $\mathbb{L}_I$ and $\mathbb{E}_I$ the set of states and the set of events in the internal domain knowledge.

Both the internal behavior of a domain object, as well as the fragments it provides to others, are modeled as processes. A process is a state transition system, where each transition corresponds to a process activity. In particular, we distinguish four kind of activities: *input* and *output* activities model communications among domain objects; *concrete* activities model internal operations; and *abstract* activities correspond to abstract tasks to be refined at run-time. All activities can be annotated with preconditions and effects, while abstract activities are annotated also with goals. For instance, let consider the example of fragment shown in Fig. 5: input/output activities are represented with an entering/outgoing message; abstract activities are drawn with a dotted line, while concrete activities are defined by solid lines. We define a *process* as follows:

Definition 5 (Process) *A process defined over an internal domain knowledge* $\mathbb{DK}_I$ *and an external domain knowledge* $\mathbb{DK}_E$ *is a tuple* $p = \langle S, S_0, A, T, Ann \rangle$*, 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;*
- $Ann = \langle Pre, Eff, Goal \rangle$ *is a process annotation, where* $Pre : A \rightarrow 2^{\mathbb{L}_I} \cup 2^{\mathbb{L}_E}$ *is the precondition labeling function,* $Eff : A \rightarrow 2^{\mathbb{E}_I} \cup 2^{\mathbb{E}_E}$ *is the effect labeling function, and* $Goal : A_{abs} \rightarrow 2^{\mathbb{L}_E}$ *is the goal labeling function.*

We denote with $S(p)$, $A(p)$, and so on, the corresponding elements of p.

We say that the precondition of the activity a is satisfied in the domain knowledge state $\overline{l} \in \mathbb{L}_I \cup \mathbb{L}_E$, and denote it with $\overline{l} \models Pre(a)$, if $\overline{l} \in Pre(a)$. Similarly, we say that the goal of the activity a is satisfied in $\overline{l} \in \mathbb{L}_I \cup \mathbb{L}_E$, and denote it with $\overline{l} \models Goal(a)$, if $\overline{l} \in Goal(a)$. Notice that the goal of an abstract activity specifies a subset of states in the external domain knowledge. As mentioned earlier, a goal can thus effectively be seen as a *disjunction of conjunctions* of states of domain properties. We say that the effects of activity a are applicable in the domain knowledge state $\overline{l} \in \mathbb{L}_I \cup \mathbb{L}_E$, if for each event $e \in Eff(a)$ there exists a $dp_i \in \mathbb{DK}$ and $l'_i \in L(dp_i)$ such that $(\overline{l}\downarrow_{dp_i}, e, l'_i) \in T(dp_i)$.

In particular, in our approach, processes are modeled as Adaptable Pervasive Flows (APF) that is an extension of traditional work-flow languages making processes suitable for adaptation and execution in dynamic environments.

Definition 6 (Domain Object) *A domain object is a tuple* $o = \langle \mathbb{DK}_I, \mathbb{DK}_E, p, \mathbb{F} \rangle$*, where:*

- $\mathbb{DK}_I$ *is an internal domain knowledge,*
- $\mathbb{DK}_E$ *is an external domain knowledge,*
- p *is a process, called core process, defined on* $\mathbb{DK}_I$ *and* $\mathbb{DK}_E$*,*
- $\mathbb{F} = \{f_1, \ldots, f_n\}$ *is a set of processes, called fragments, defined on* $\mathbb{DK}_I$ *and* $\mathbb{DK}_E$*, where for each* $f_i \in F$*,* $a \in A_{in}(f_i)$ *implies* $a \in A_{out}(p)$ *and* $a \in A_{out}(f_i)$ *implies* $a \in A_{in}(p)$*.*

The latter constraint on fragments specification concerns the fact that input/output activities in fragments represent explicit communication with the provider domain object. Thus fragments, once received by other domain objects and injected in their own process, start a peer-to-peer communication with the core process of the provider, that implements the required functionality. A graphical representation of a domain object is reported in Fig. 2.

Definition 7 (Adaptive System) *An adaptive system is modeled as a set of domain objects:* $AS = \{o_1, \ldots, o_n\}$*.*

Figure 3, for instance, shows a portion of the travel assistant adaptive system. In it, we say that there is a *soft dependency* between objects o_1 and o_2, denoted with $o_1 \dashleftarrow o_2$, if o_1 requires a functionality that is provided by o_2. A soft dependency is formally defined as follows:

Definition 8 (Soft Dependency) $\forall o_i, o_j \in AS$ *with* $o_i \neq o_j$*,* $o_i \dashleftarrow o_j$ *if there exists* $dp_E \in \mathbb{DK}_E(o_i)$ *then there exists* $dp_I \in \mathbb{DK}_I(o_j)$ *such that* $dp_E = dp_I$*.*

In the next section we introduce the adaptation mechanisms and strategies *exploited* and *facilitated* by our design for adaptation approach, as well as the enablers for the execution and adaptation of service-based applications.

5 Adaptive service-based applications: execution

In this section, we first provide an overview on the adaptation mechanisms and strategies exploited by our approach [22], in Section 5.1. In Section 5.2, we give a description of the enablers of the design for adaptation. For illustration purpose, we provided a running scenario of the travel assistant example in Section 5.3. The execution model is formalized in Section 5.4.

5.1 Adaptation mechanisms and strategies

The adaptation mechanisms and strategies that we employ implement the dynamic adaptation of fragment-based and context-aware business processes proposed in [22], which are in turn based on AI planning [23]. The link between the approach presented in this article and the approaches in [22] is the use of the APFL to model processes. It allows developers to define flexible processes that are particularly suitable for *adaptation* and *execution* in dynamic environments.

The used adaptation mechanisms deal with three types of adaptation needs. The first, which is the one we mainly use in our scenario, refers to the need for refining an abstract activity. This is made by triggering the ***refinement mechanism*** whose execution allows the approach to automatically find and compose available fragments in the application, on the basis of the goal of the abstract activity and the current context. As a result, an executable process whose execution guarantees to reach the abstract activity's goal is provided (details are given in Section 5.3).

The second is called ***local adaptation mechanism*** and it refers to the violation of the precondition of an activity that has to be performed. It requires for a solution helping in re-starting a faulted process. For instance, booking a place in a ride-share is constrained by a precondition requiring that the user is subscribed to the specific ride-sharing service.

The last is called ***compensation mechanism*** and it allows designers to avoid the explicit definition of activities' compensation procedures, and to dynamically provide a context-aware compensation process (i.e., when a travel ticket refund is needed).

Furthermore, the AI planning on which the goal-based adaptation relies is able to deal with *stateful* and *non-deterministic* services. In addition, the fragments composition (i.e., a plan) returned by the AI planner as a result to an adaptation problem is correct by construction [23], that is, if a plan is found, it is guaranteed that its execution allows the application to reach a situation in which the goal of the adaptation problem is satisfied. However, dealing with stateful services implies that the planner might even not find a solution to an adaptation problem. For these reasons, adaptation strategies have been designed. Indeed, the mechanisms introduced above can be further combined into *adaptation strategies* allowing the application to handle more complex adaptation needs (e.g., the failure of an abstract activity refinement). The before-mentioned mechanisms and strategies have all been implemented in an **adaptation engine** [24]. This engine is *one* of the enablers of our design for adaptation approach.

5.2 Enablers of the design for adaptation approach

The run-time operation of service-based applications realized with our approach relies on different *execution and adaptation enablers*, shown in Fig. 7.

The **Execution Enablers**, namely the *Domain Objects Manager* and the *Process Engine*, leverage on the different services wrapped up as domain objects and stored in the application's knowledge base. The execution enablers are in charge of executing the domain objects processes (i.e., core processes and fragments) during the operation of service-based applications. The **Adaptation Enablers**, namely the *Refinement Handler*, the *Adaptation Manager* and the *AI planner*, instead, leverage on the adaptation mechanisms and strategies, described in Section 5.1. They are in charge of managing the adaptation needs of applications, arising at run-time. Consided as a whole, they represent the *adaptation engine*.

To start, it is required that developers select the available services in a given domain (e.g., mobility) and wrap-up them as domain objects. These are stored in the *Domain Objects Models* repository in Fig. 7. To understand how the execution and adaptation enablers interact, we defined a sequence diagram in Fig. 8.

Domain objects core processes (simply processes from here on) are executed by the *Process Engine*. It manages service requests among processes and, when needed, it sends requests for domain objects instantiation to the *Domain Objects Manager*. A request is sent for each demanded service whose corresponding process has not yet been instantiated. The domain objects manager replies by deploying the requested process on the process engine. In this way, a correlation between the two processes is defined.

During the normal execution of processes, abstract activities can be met. These activities need to be refined with one or a composition of fragments modeling services functionalities. To this aim, the process engine sends a request for abstract activity refinement to the *Refinement Handler* component. This component is in charge of defining the *adaptation problem* corresponding to the received request. In particular, the *adaptation problem* is represented by: (i) a set of fragments that can potentially be part of the final fragments composition. The selection is driven by the goal defined by the abstract activity. (ii) A set of domain properties, and (iii) the adaptation goal. The *planning domain* is then derived from the adaptation problem by transforming fragments and domain properties into STS, by applying transformation rules, such as those presented in [25]. The adaptation goal is, instead, transformed into a set of configuration of the planning domain. Then, the refinement handler submits the adaptation problem to the *Adaptation Manager*. This translates the *adaptation problem* into a *planning problem* so that it can be solved by the *AI Planner* component. After the plan generation (i.e., made as a STS), the AI planner sends the plan to the adaptation manager that will transform it into an executable process. This process can now be sent to the process engine and injected into the abstract activity being refined. At this point, depending on the fragments in the composition, the process engine can request for the instantiation of one or more domain objects, whose processes will be deployed. At the end, the execution of the refinement process can be performed.

5.3 Travel assistant: running scenario

In this section, we show a concrete example on the running execution of the travel assistant. The focus of this section is that of showing (i) how domain objects dynamically inter-operate by exchanging and injecting (composition of) fragments, thus enabling a *chain of incremental refinements* (such as that in Fig. 9); (ii) how *the refinement process allows domain objects to span their external knowledge on the domain*, by establishing new soft dependency.

The main features of the travel assistant are the following: **(i)** collect the user's requirements (e.g., source and

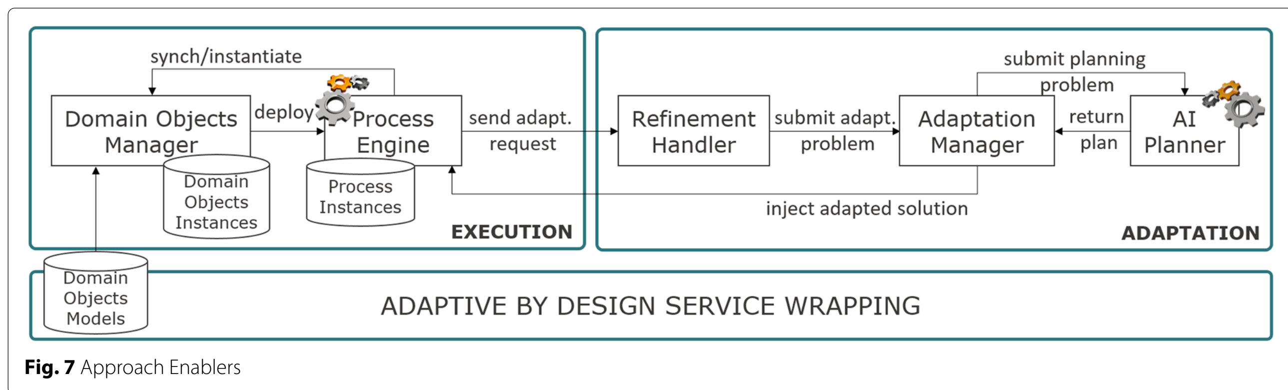

Fig. 7 Approach Enablers

destination points, travel preferences, etc) and set up a journey planning request; **(ii)** run a local or a global planning; **(iii)** identify the transport means in the journey's legs of the solution selected by the user. This way, it goes vertically to find the proper service(s) to use (e.g., the ones of the specific transport companies), if existing in the application.

Executing the travel assistant. Our user, Sara, wants to organize a journey from Trento to Vienna. In Fig. 9, we report examples of *chains of incremental refinements*, as they are dynamically set up and executed after the specific request of Sara[4].

The travel assistant is provided as a mobile application (modeled by the domain object `Travel Assistant Application` in Fig. 3), through which Sara uses it. The execution starts from the core process of this mobile app, modeling the user process. Then, a sequence of three abstract activities (represented with dotted lines and labeled with a goal) need to be refined (see the top side of Fig. 9). Here we focus on the first one, `Plan Journey`, whose goal models the situation in which Sara ends up with a specific travel plan. The refinement mechanism is triggered and the following steps are performed (see Fig. 9).

Step 1. The fragment `PlanJourney` provided by the Travel Assistant is selected for the refinement, and injected in the behavior of the mobile app core process. It implements a wider journey planning functionality, allowing for looking for available alternatives and performing a more detailed planning after that a specific alternative has been found and selected by the user. To start, it allows Sara to insert the departure and destination locations.

Step 2. To identify the proper planning mode (*local* vs. *global*), the travel assistant domain object relies on the `Planners Management` domain property, as shown by the abstract activity `Travel Assistant Plan Journey` in the `PlanJourney` fragment in execution. The Journey Planners Manager domain object implements the `Planners Management` domain property. Its fragment `SelectPlanningMode` is selected for the refinement. This fragment does not implement any logic. Indeed, its activities `Plan Request` and `Receive Planning Type` model the communication with its core process, where the request is effectively handled. In particular, the Journey Planners Manager knows only at runtime if a *local* or *global* planning is required. In our scenario, having Trento and Vienna, the Journey Planners Manager will reply with a global planning type. This will drive the execution of its fragment through the `Plan Global Journey` abstract activity.

Step 3. At this point, one or more fragments provided by the available global journey planners existing as domain objects in the application's knowledge base can be selected for the refinement. In our scenario, we suppose that the `Plan Global Journey` abstract activity is refined with the fragment `Plan` provided by the Rome2Rio[5] domain object, a open global planner service. The execution of this fragment will end up with a list of travel alternatives, if any.

Step 4. After that the chain of incremental refinements made by the steps 1, 2 and 3 has been accomplished, the execution returns to the `PlanJourney` fragment, by continuing with the `DataViewerPattern` abstract activity. Indeed, an appropriate data visualization pattern must be selected, based on the data format (e.g., a list, a message). This is defined at run-time, when the data (and its format) is known. The Data Viewer domain object provides the `DefineDataViewer Pattern` fragment for this purpose. At this point, Sara can receive and visualize on her smartphone the list of the found travel alternatives satisfying her requirements.

Step 5. Sara can now select her preferred alternative (we suppose that she selects a multi-modal solution made

[4]We remark that more complex alternatives of our scenario can be modeled within our approach. In this article we use a trivial but exhaustive example to highlight the features of the approach. For presentation purposes and without loss of generality, we report only portions of the processes involved in the scenario. For each fragment, we specify its name and the domain object which it belongs to (e.g., fragmentName@domainObjectName).

[5]https://www.rome2rio.com

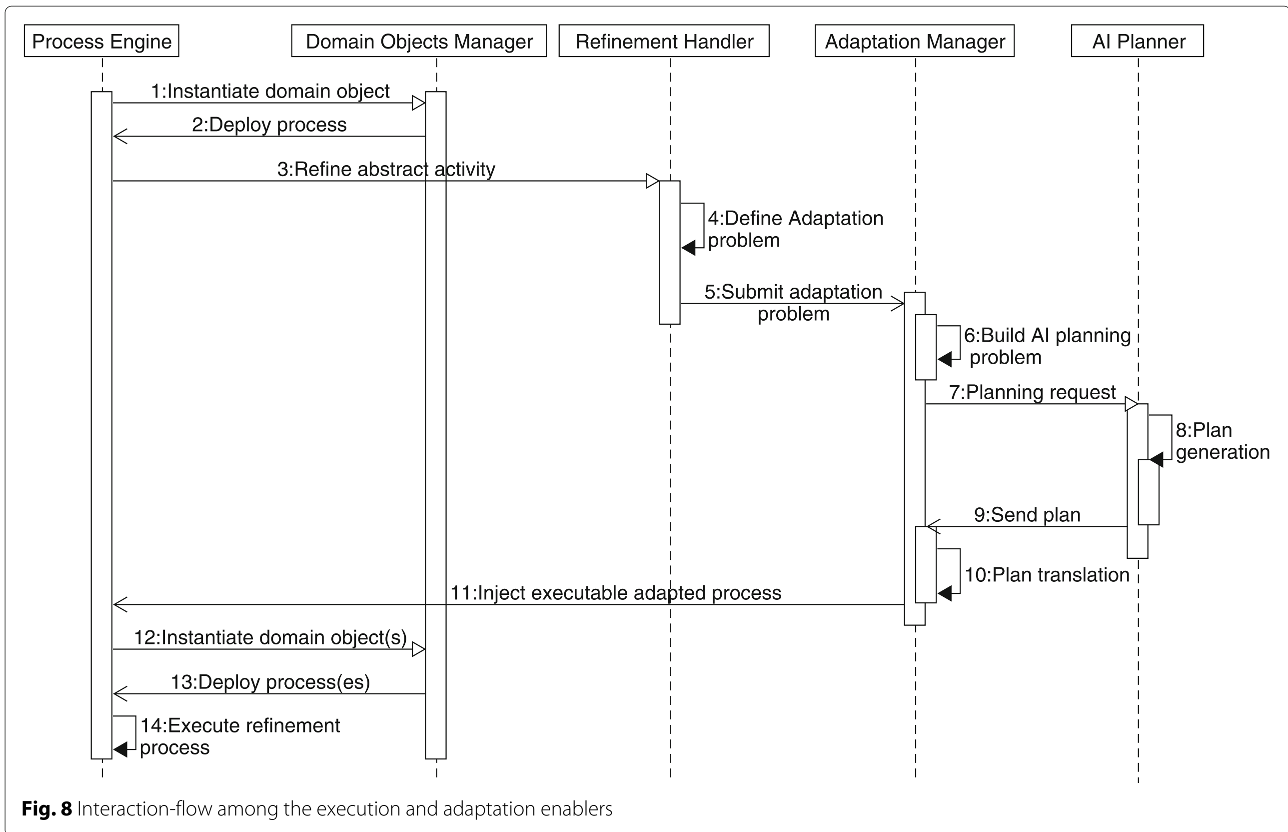

Fig. 8 Interaction-flow among the execution and adaptation enablers

by a train and a bus travels). Based on the user choice, the `Define Journey Legs` abstract activity is refined with the `HandleJourneyLegs` fragment provided by the Journey Manager domain object. It is able to dynamically define the goal for the `Refine Journey` abstract activity, that will be `G: TJ = Response Sent AND BJ = Response Sent`, being the selected solutions made by a train and a bus journeys. The refinement of this abstract activity will allow the Travel Assistant to look for and find the proper fragments for each journey leg. Notice that the `Refine Journey` activity is a so-called *higher order abstract activity*, that we are going to define in the subsequent paragraph.

Step 6. The last step shows a *composition of fragments* provided by the transport companies involved in the legs of the journey alternative selected by the user (i.e., *Sudtirol Alto Adige* and *Hello*). Their execution provides to Sara the proper solutions, from the two companies, that combined together satisfy her need of planning a journey from Trento to Vienna, passing through Bozen.

Higher Order Abstract Activities. In step 5 of the running example, we have presented the `Refine Journey` activity as a *Higher Order Abstract Activity* (HOAA). This kind of activity is actually a regular abstract activity and it is managed as such, with the only difference that its goal is defined at execution time, within the fragment or core process it belongs to. For instance, in Fig. 9 – Step 5, we can notice that the `Receive Goal for Legs Specialization` activity, is in charge of receiving the HOAA's goal and labeling the `Refine Journey` HOAA with it, so that, at the next step, the process engine can execute it.

HOAAs are used for those abstract activities whose goal's specification is *fully* depending from the run-time execution environment. Specifying such a goal (i.e., a composition requirements) at design time, would mean defining all the possible alternatives that the goal could assume. But this is exactly what must be avoided. For this reason, we introduced the HOAA construct allowing for the dynamic definition of goals when the execution domain is known.

Example 8 *The* `HandleJourneyLegs` *fragment exploited at Step 5 in the running example is exposed by the Journey Manager domain object. Its main task is that of relating a specific travel alternative selected by a user with the proper domain objects able to handle it. It is easy to notice that a travel solution can be made from any possible combination of transport means. This implies that the goal of the* `Refine Journey` *HOAA, if defined*

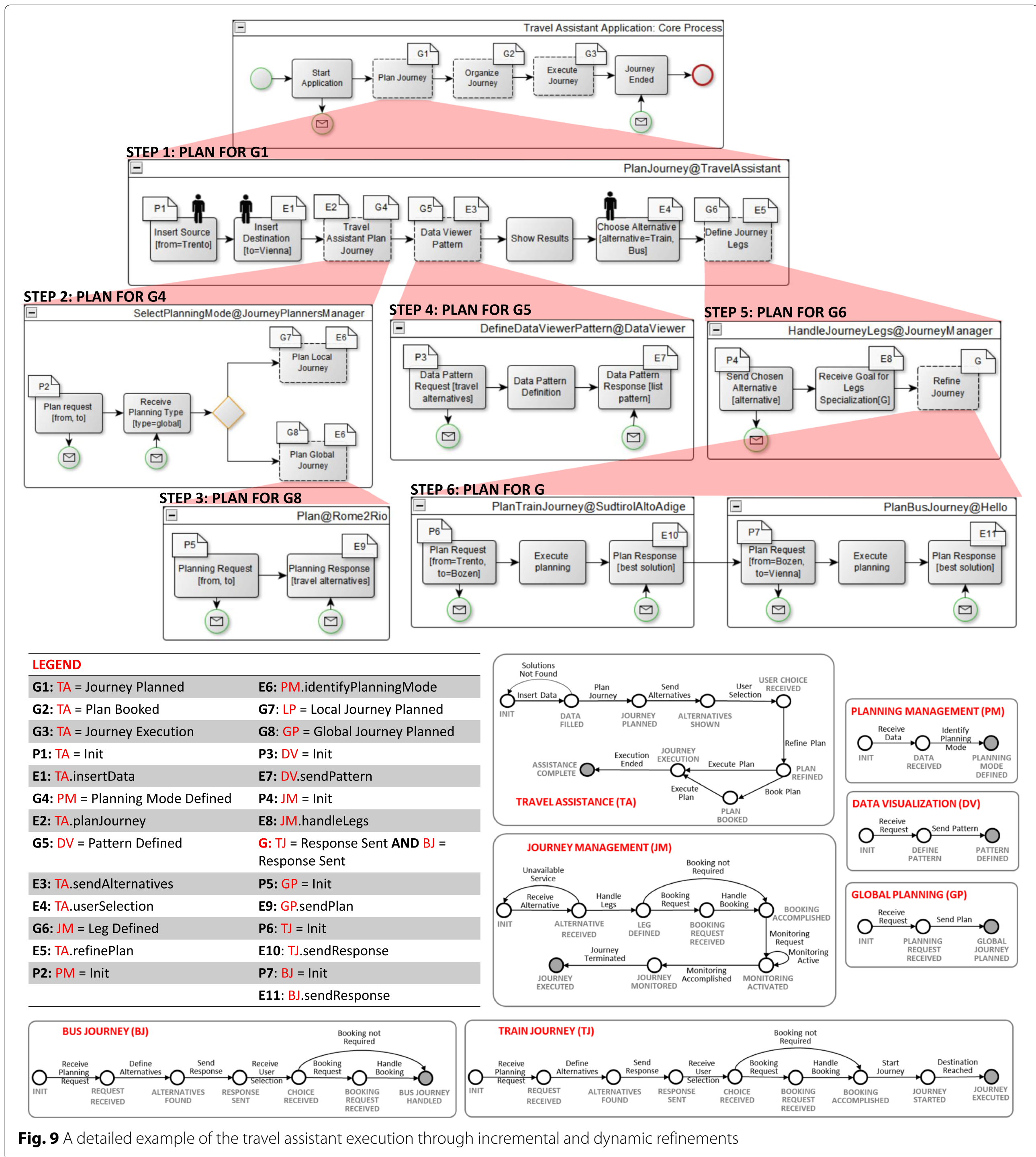

Fig. 9 A detailed example of the travel assistant execution through incremental and dynamic refinements

at design time, should model any possible configuration to cover all the corresponding combination of transport means. To the contrary, the Journey Manager implements the logic to dynamically relate a combination of transport means (e.g., train and bus as in our example) with the right goal to be associated with the HOAA handling it (e.g., the goal `G: TJ = Response Sent AND BJ = Response Sent` *in Fig. 9), which is dynamically generated.*

5.3.1 *Dynamic knowledge extension*

An important feature of our approach is represented by the ability of domain objects to span their knowledge on the whole application domain.

The *dynamic extension of the knowledge* concerns with the *external domain knowledge* and it is triggered by the execution of the abstract activity refinement mechanism. In particular, it takes place every time that a domain object injects in its own core process one or more fragments containing abstract activities. Indeed, since abstract activities are labeled with a goal, the receiving domain object receives, together with the fragments, also those domain properties on which fragments execution rely on. These domain properties will extend the external domain knowledge.

For instance, in Fig. 10 we depicted the evolution of the external domain knowledge in the `Travel Assistant Application` domain object, after the execution of step 1 and step 2 of Fig. 9. Both steps, indeed, are characterized by the injection of fragments, namely `PlanJourney` and `SelectPlanningMode`, equipped with abstract activities, whose goals (i.e., `G4, G7, G8` – see table in Fig. 9) rely on domain properties which are automatically inherited by the `Travel Assistant Application` domain object. More specifically, the `Planning Management`, `Local Planning` and `Global Planning` properties are received.

This dinamicity is now reflected in the soft dependencies of the Travel Assistant Application because new dependencies are established. In particular, it will establish new dependencies with all the domain objects in the application implementing the three just inherited domain properties.

We can notice how the dynamic knowledge extension allows domain objects to dynamically discover new services that they can, in turn, exploit for the refinement of inherited abstract activities. It is easy to note that the refinement at the step 3 in Fig. 9 would not have been possible without the dynamic extension of the knowledge because, in its design time version, the Travel Assistant Application did not have the `Global Planning` knowledge required to do it. Lastly, we want to highlight that if new global planners enter the application, the Travel Assistant Application will be able to know and exploit them in its further execution, thanks to the establishment of new soft dependencies.

5.4 Execution model formalization

The following definition captures the current status of the execution of a given core process. The process instance is a hierarchical structure, obtained through the refinement of abstract activities into fragments. A process instance is hence modeled as a list of tuples process-activity: the first element in the list describes the fragment currently under execution and the current activity; the other tuples describe the hierarchy of ancestor fragments, each one with abstract activities currently under execution. The last element in the list is the process model from which the running instance has been created. A process instance is defined as follows:

Definition 9 (Process Instance) *We define a process instance I_p of a process p as a non-empty list of tuples $I_p = (p_1, a_1), (p_2, a_2), \ldots, (p_n, a_n)$, where:*

- *each p_i is a process and $p_n = p$;*
- *$a_i \in A(p_i)$ are activities in the corresponding processes, with $a_i \in A_{abs}(p_i)$ for $i \geq 2$ (i.e., all activities that are refined are abstract).*

An example of process instance is given by the process of the Travel Assistant Application domain object, shown in Fig. 9, where we reported an example of its execution. A domain object instance, instead, is specified as follows.

Definition 10 (Domain Object Instance) *A domain object instance δ of a domain object $o = \langle \mathbb{DK}_I, \mathbb{DK}_E, p, \mathbb{F} \rangle$ is a tuple $\delta = \langle \mathbb{DK}_I, \mathbb{DK}_{E^+}, \bar{l}_I, \bar{l}_{E^+}, I_p \rangle$ where:*

- *$\mathbb{DK}_{E^+} \supseteq \mathbb{DK}_E$, is the current set of domain properties in the external domain knowledge;*
- *$\bar{l}_I \in \mathbb{L}_{DK_I}$ and $\bar{l}_{E^+} \in \mathbb{L}_{DK_{E^+}}$ are the current state of the domain properties in the internal and external domain knowledge;*
- *I_p is its process instance.*

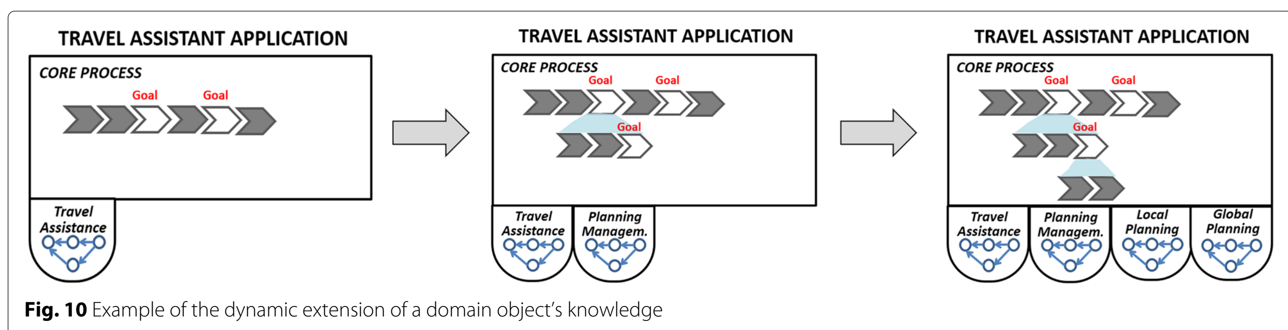

Fig. 10 Example of the dynamic extension of a domain object's knowledge

Notice that $\mathbb{DK}_{E^+} = \mathbb{DK}_E$ when the domain object is instantiated. Then, $\mathbb{DK}_{E^+}$ might grow during the domain object execution; this mechanism is formally defined later on.

We define now an adaptive system instance.

Definition 11 (Adaptive System Instance) *An adaptive system instance AS_I of an adaptive system $AS = \{o_1, \ldots, o_n\}$ is a set of domain object instances $AS_I = \{\delta_{ij}\}$ where each δ_{ij} is an instance of domain object o_i.*

For instance, if we consider the running scenario depicted in Fig. 9 of the travel assistant system, we can say that the adaptive system instance, for that specific execution, is made by instances of the `Travel Assistant Application`, `Travel Assistant`, `Journey Planners Manager`, `Journey Manager`, `Data Viewer`, `Rome2Rio`, `Train` and `Bus` domain objects.

We will now formally define the execution model of domain objects. In the following a refinement need is formalized.

Definition 12 (Refinement need) *A refinement need is a tuple $\eta = \langle AS_I, \delta, a \rangle$ where:*

- *AS_I is an adaptive system instance;*
- *$\delta \in AS_I$ is the domain object instance for which the refinement is needed;*
- *a is the abstract activity of δ to be refined.*

For instance, considering the process whose refinement is shown in Fig. 9, the domain object instance for which the refinement is needed is an instance of the Travel Assistant Application, while the abstract activity to be refined is the `Plan Journey` activity.

A refinement is defined as follows.

Definition 13 (Refinement) *A refinement for a refinement need $\eta = \langle AS_I, \delta, a \rangle$, denoted with $REF(\eta)$, is a tuple $\langle p_\eta, \mathbb{DK}_\eta, \bar{l}_\eta \rangle$ where:*

- *p_η is the process to be injected;*
- *$\mathbb{DK}_\eta$ is the set of domain properties to be added to the external domain knowledge;*
- *for each $a \in A_{abs}(p_\eta)$, $Goal(a) \subseteq 2^{\mathbb{L}_{DK_\eta}}$;*
- *$\bar{l}_\eta \in \mathbb{L}_{DK_\eta}$ is the current state of the domain properties.*

The last two items of the previous definition require that, in case the refinement process contains abstract activities, the domain knowledge needed for their refinement is part of the refinement solution. Indeed, this is how the domain knowledge extension is performed.

We will now characterize a correct solution for a refinement need η. Intuitively, a refinement $\langle p, \mathbb{DK}, \bar{l} \rangle$ is a correct solution to a refinement need $\eta = \langle AS_I, \delta, a \rangle$, if the execution of p brings the external domain knowledge of object δ in a state that satisfies the goal of a. Notice that p, being a composition of fragments provided by other domain objects, might contain abstract activities that will be refined later on, when the refinement is executed. Our definition of correct refinement is based on the assumption that abstract activities, once refined, will behave as declared in their specification (preconditions and effects on their activities). That is, we treat them as all other activities in the process, assuming that their behavior is correctly specified through their annotations in terms of preconditions and effects.

In the following we give the definitions of action executability, action impact, and abstract run of a process. These definitions are the basis for the formal characterization of a correct refinement.

Definition 14 (Action Executability) *An action a of a process p is executable from domain knowledge state $\bar{l} \in \mathbb{L}_{DK}$, denoted with $Executable(a, \bar{l})$, if $\bar{l} \models Pre(a)$ and the effects of action a are applicable in domain knowledge state $\bar{l}$.*

In other words, an action is executable from a given domain knowledge state if, in that state, its precondition is verified and its effects can be applied.

Definition 15 (Action Impact) *The impact of action a belonging to some process p when executed from domain knowledge state $\bar{l} \in \mathbb{L}_{DK}$, denoted with $Impact(a, \bar{l})$, is a domain configuration $\bar{l}' \in \mathbb{L}_{DK}$ such that for every $dp_i = \langle L_i, l^0{}_i, E_i, T_i \rangle \in \mathbb{DK}$, if exists an $e \in Eff(a)$ such that $\left(\bar{l}\downarrow_{dp_i}, e, l'_i\right) \in T_i$ then $\bar{l}'\downarrow_{dp_i} = l'_i$, otherwise $\bar{l}'\downarrow_{dp_i} = \bar{l}\downarrow_{dp_i}$.*

The action impact is given by the domain configuration in which the domain knowledge of the domain object that is executing the activity evolves.

Definition 16 (Abstract Process Run) *Given a process $p = \langle S, S_0, A, T, Ann \rangle$ and a domain knowledge state $\bar{l} \in \mathbb{L}_{DK}$, $\pi = (s_1, a_1, s_2, \ldots, a_{n-1}, s_n)$ is an abstract run of p from $\bar{l}$ if:*

- *$s_1 \in S_0$ and $\forall i, \in [1, n] : s_i \in S$;*
- *$\forall i \in [1, n-1] : a_i \in A$ and $(s_i, a_i, s_{i+1}) \in T$;*
- *there exists a domain knowledge evolution of $\mathbb{DK}$, $\pi_{DK} = (\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_n)$ such that:*
  - *$\bar{l}_1 = \bar{l}$;*
  - *$Impact(a_i, \bar{l}_i) = \bar{l}_{i+1}$ for all $i \in [1, n-1]$;*
  - *$Executable(a_i, \bar{l}_i)$ for all $i \in [1, n-1]$.*

A process run that terminates in a state with no outgoing transitions (final state) is called a complete run. We denote with $\Pi_{ABS}(p, \bar{l})$ the set of all possible complete abstract runs of process p from domain knowledge state $\bar{l} \in \mathbb{L}_{DK}$.

We can now define a correct refinement.

Definition 17 (Correct Refinement) *Given a refinement need* $\eta = \langle AS_I, \delta, a \rangle$*, with* $\delta = \langle \mathbb{DK}_I, \mathbb{DK}_{E^+}, \bar{l}_I, \bar{l}_{E^+}, I_p \rangle$*, we say that a refinement* $\langle p_\eta, \mathbb{DK}_\eta, \bar{l}_\eta \rangle$ *is a correct solution for* η*, if for each complete abstract run* $\pi \in \Pi_{ABS}(p_\eta, \bar{l}_{E^+})$*, its associated domain knowledge evolution* $\pi_{DK} = (\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_n)$ *is such that* $\bar{l}_n \models Goal(a)$*.*

Intuitively, a refinement is a correct solution for a refinement need if all its complete abstract runs satisfy the goal of the abstract activity to be refined.

As regards the execution of an adaptive system instance, intuitively, it evolves in three different ways. First, through the execution of activities in domain object instances, which will be presented in detail in the following. Second, through the interaction among domain object instances, which happens according to the standard rules of peer-to-peer process communication. Third, through a change in the behavior, or entrance / exit, of domain objects and domain object instances into the system.

In the following we formalize the execution model of a domain object, considering also the injection of a refinement solution in the case in which an abstract activity is executed.

Definition 18 (Action Execution) *Given a domain object instance* $\delta = \langle \mathbb{DK}_I, \mathbb{DK}_{E^+}, \bar{l}_I, \bar{l}_{E^+}, I_p \rangle$*, with* $\delta \in AS_I$ *and* $I_p = (p_1, a_1), (p_2, a_2), \ldots,$ (p_n, a_n)*, the execution of action* a_1*, denoted with* $exec(\delta, AS_I)$*, evolves* δ *to* $\langle \mathbb{DK}_I, \mathbb{DK}'_{E^+}, \bar{l}'_I, \bar{l}'_{E^+}, I_p \rangle$*, where:*

- *if* $a_1 \in A_{in}(p_1) \cup A_{out}(p_1) \cup A_{con}(p_1)$ *then*
  - $\mathbb{DK}'_{E^+} = \mathbb{DK}_{E^+}$;
  - $\bar{l}'_I = Impact(a, \bar{l}_I)$ and $\bar{l}'_{E^+} = Impact(a, \bar{l}_{E^+})$;
  - *if* $next(p_1, a_1) \neq \emptyset$ *then* $I'_p = (p_1, next(p_1, a_1)), (p_2, a_2), \ldots, (p_n, a_n)$, *otherwise* $I'_p = (p_2, next(p_2, a_2)), \ldots, (p_n, a_n)$.
- *if* $a_1 \in A_{abs}(p_1)$*, given* $\langle p_\eta, \mathbb{DK}_\eta, \bar{l}_\eta \rangle = REF(\eta)$*, with* $\eta = \langle AS_I, \delta \rangle$*, then*
  - $\mathbb{DK}'_{E^+} = \mathbb{DK}_{E^+} \cup \mathbb{DK}_\eta$;
  - $\bar{l}'_{E^+} \in \mathbb{L}_{DK'_{E^+}}$ *is such that for every* $dp_i = \langle L_i, l^0{}_i, E_i, T_i \rangle \in \mathbb{DK}'_{E^+}$*, if* $dp_i \in \mathbb{DK}_\eta$ *then* $\bar{l}'_{E^+}{\downarrow}_{dp_i} = \bar{l}_\eta{\downarrow}_{dp_i}$*, otherwise* $\bar{l}'_{E^+}{\downarrow}_{dp_i} = \bar{l}_{E^+}{\downarrow}_{dp_i}$*;*
  - $I'_p = (p_\eta, a^0_\eta)(p_1, a_1), (p_2, a_2), \ldots, (p_n, a_n)$.

Eventually, we previously said as a soft dependency among two domain objects becomes a strong dependency, denoted with $\delta_{ih} \leftarrow \delta_{jk}$, if the domain object δ_{ih} injects in its internal process a fragment provided by δ_{jk}. This is formally defined as follows:

Definition 19 (Strong Dependency) $\forall \delta_{ih}, \delta_{jk} \in AS_I$ *with* $i \neq j$ *and* $h \neq k$*,* $\delta_{ih} \leftarrow \delta_{jk}$ *if* $\exists (f, a) \in I_p(\delta_{ih}) | f \in \mathbb{F}(o_j)$*.*

In the next section, we show how the refinement problem previously presented can be solved by applying the automated fragment composition approach based on AI planning [22].

5.4.1 Automated refinement via AI planning

Within the approach presented in [23] and summarized in Section 7, we said that a fragment composition problem is transformed into a planning problem. Relevantly to our purposes, such techniques cover uncertainty, in order to allow the composition of services whose dynamics is only partially exposed, and is able to deal with complex goals and data flow [25].

In the following we briefly describe how a refinement need $\eta = \langle AS_I, \delta, a \rangle$, with $\delta = \langle \mathbb{DK}_I, \mathbb{DK}_{E^+}, \bar{l}_I, \bar{l}_{E^+}, I_p \rangle$ is transformed into an AI planning problem. In other words, we say how the approach in [23] is adjusted and used in our framework.

First of all, a set of n fragments, $(f_1, \ldots, f_n)$, is selected from the soft dependencies of δ: for some $\delta' \in AS_I$, with $\delta \dashleftarrow \delta', f_i \in \mathbb{F}(\delta')$.

Advanced optimization techniques, as the one described in [26], can be used to further reduce the set of fragments on the basis of the functionalities they provide and of the preconditions satisfiability of their preconditions in current domain knowledge state. Both fragments $(f_1, \ldots, f_n)$ and the set of domain properties $(dp_1, \ldots, dp_m) \in \mathbb{DK}^+_E$, on which the fragments are annotated, are transformed into state transition systems (STS) using transformation rules similar to those presented in [23]. During this encoding, all goals on abstract activities in fragments are ignored, while preconditions and effects are maintained. With this measure, the refinement plan will be built under the assumption that abstract activities will behave according to their annotation, independently from the way in which they will be refined (see Definition 17).

The planning domain Σ is obtained as the product of the STSs $\Sigma_{f_1} \ldots \Sigma_{f_n}$ and $\Sigma_{dp_1} \ldots \Sigma_{dp_m}$, where STSs of fragments and domain properties are synchronized on preconditions and effects, $\Sigma = \Sigma_{f_1} \| \ldots \| \Sigma_{f_n} \| \Sigma_{dp_1} \| \ldots \| \Sigma_{dp_m}$. The initial state of the planning domain is derived from the initial state of all fragments and the current state of the domain properties $\bar{l}_{E^+}$, by interpreting it as states of the STSs defining the planning domain.

Similarly, the refinement goal $Goal(a)$ is transformed into a planning goal ρ by interpreting the states in $\mathbb{DK}_{E^+}$ as states in the planning domain.

Finally, the approach of [23] is applied to domain Σ and planning goal ρ to generate a plan Σ_η that guarantees achieving goal ρ once *executed* on system Σ. State transition system Σ_η can be further translated into an executable process p_η, which implements the identified solution.

6 Prototype implementation and validation

This section is devoted to the architecture of the design for adaptation approach, and the implementation and evaluation of an application on top of it, namely *ATLAS*, which implements the travel assistant scenario. The aim of this section is to demonstrate the feasibility of the approach for realizing adaptive applications.

6.1 Design for adaptation architecture

From a technical perspective, the architecture is organized in three main layers, as shown in Fig. 11.

The **Enablers** leverage on our results on the design for adaptation approach, described in Section 4. Developers can exploit and wrap up as domain objects the available services in the target domain. Besides the design of services, *execution* and *adaptation* enablers allow also for their run-time operation, as described in Section 5.2. Moreover, to deal with IoT domains, or more generally with IoT things, the *IoT Platform Services* has been added, together with the *Things States* repository. The former can relate to any cloud platform providing IoT services (e.g., Amazon AWS-IoT platform[6]) enabling the management and interaction with things. The Process Engine can send instructions to things through the IoT Platform Services component (e.g., when executing activities including calls to things API). The Domain Objects Manager is responsible for answering queries about available IoT things and their capabilities. The latter, stores knowledge about things operational states.

The **Provided Services** layer exposes the functionalities implemented by the Enablers. These services can exploit and/or combine into value-added services (e.g., a travel assistant in the mobility domain) the services previously wrapped up and made available by the Enablers. The key idea is that the architecture is open to continuous extensions with new services, wrapped as domain objects, whose functionalities can be exploited in a transparent way to provide value-added services to the end-users.

All the provided services can be eventually delivered to final users through a range of multi-channels front-end applications that constitute the **Front-end** layer. These can be mobile or desktop applications, and they can also rely on existing services, such as chat-bots (e.g., Telegram chat-bot).

6.2 Case study: ATLAS, a smart travel assistant

In this section, we introduce ATLAS – *a world-wide personAlized TraveL AssiStant* [27]. ATLAS consists both in (i) a demonstrator showing the application's models and its execution and evolution through automatic run-time adaptation, and (ii) a Telegram chat-bot, for the interaction with the users. The demonstrator is based on a process-engine for the execution of automated and adaptable processes. Before implementing ATLAS, we looked for a process engine suitable for the integration into our design for adaptation framework, as for instance extensible with abstract activities. We come out with a subset of eligible process engines, namely *jBPM* [7], *Camunda* [8] and *Activiti* [9]. However, none of them are thought for dealing with (i) the *decentralized management of processes* and (ii) the *correlation among different processes* that are fundamental in our framework. As a consequence, we decided to realize from scratch a process engine implementing the features required by our framework. Essentially, it is a conventional process engine, extended with some adaptation-related constructs. It handles the *multiple-instance processes management*, the dynamic *correlation among processes* and the *abstract activity management*. The demonstrator also implements the enablers. In this article we focus on the chat-bot[10].

We clarify here that the implementation effort for developing ATLAS consist only in the modeling of the involved domain objects and in the realization of the dedicated Telegram chat-bot. The enablers shown in Fig. 11 are part of the design for adaptation framework and are reusable in the implementation of any other application different than ATLAS.

To realize a world-wide travel assistant we selected *real-world mobility services* exposed as open APIs. We identified their behavior, functionalities and their input and output data. Then, we wrapped them up as domain objects and stored them in the knowledge base. For instance, we wrapped *Rome2Rio* and *Google Transit*[11] as global journey planners. To overcome the limitations of global planners in terms of accuracy, we wrapped local planners too, such as *ViaggiaTrento*[12]. It can be exploited for those journey located in the city of Trento, which can also be part of a wider inter-modal travel solution provided by a global planner, but for which the global planner does not

[6] https://aws.amazon.com/iot/

[7] https://www.jbpm.org/

[8] https://camunda.org/

[9] https://www.activiti.org/

[10] To see the travel assistant in action (both demonstrator and chat-bot), or simply inspect its full specification, one can freely download ATLAS at: https://bit.ly/2V2JNy8

[11] http://www.google.com/transit

[12] http://www.smartcommunitylab.it/apps/viaggia-trento/

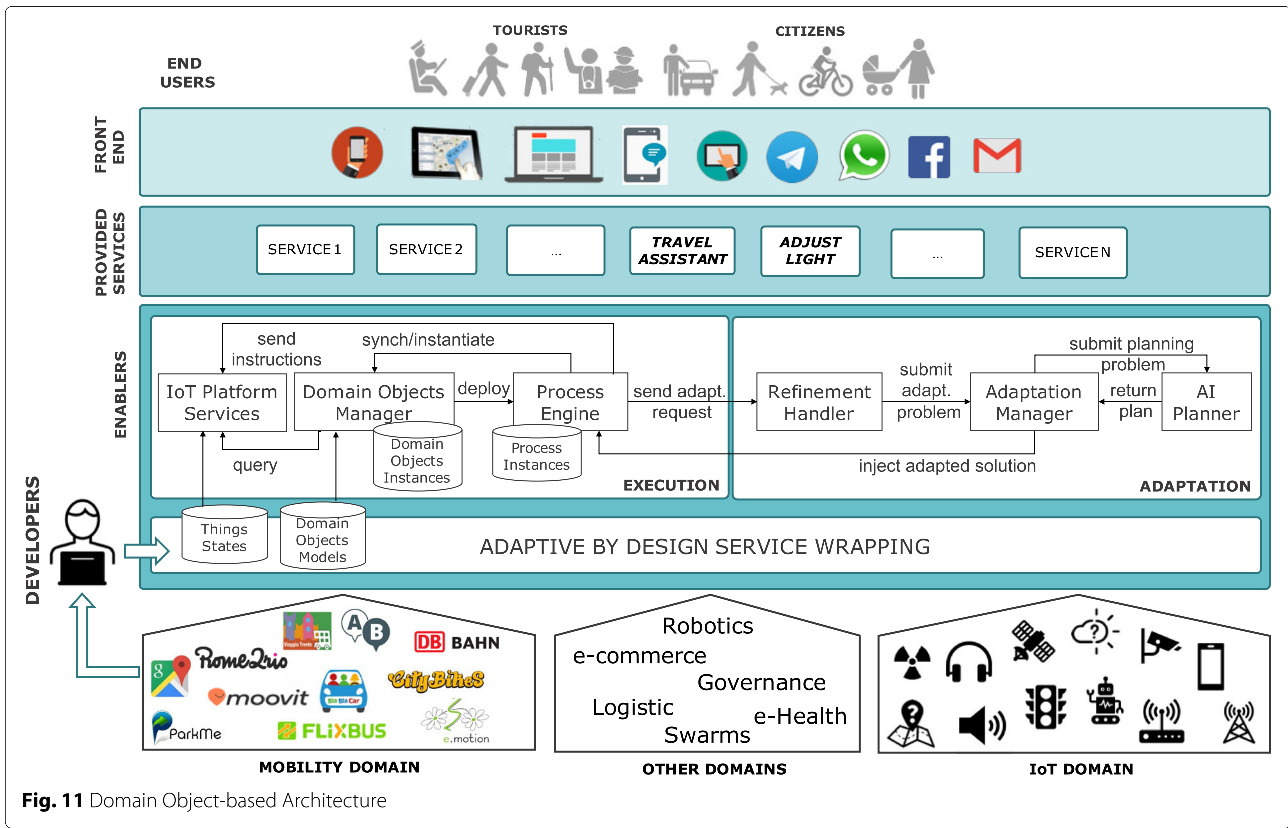

Fig. 11 Domain Object-based Architecture

give enough or accurate information. Combining the geographical coverage of global planners with the accuracy of local planners is a concrete example of services interoperability promoted by our approach. Other open mobility services we considered are *Travel for London* [13] as planner for the city of London, *BlaBlaCar* as ride sharing service, *CityBikes*[14] as bike sharing service applying to about 400 cities, to give a few examples. We emphasize that the more (mobility) services are wrapped up and stored in the application's knowledge base, the more responsive and accurate the travel assistant will be.

At the *Provided Services* level we defined the *Travel Assistant*. It has been realized as a value-added service leveraging on the services available in the application's knowledge base. Its main features have been described in Section 5.3.

Finally, among the multi-channel front-ends that can be exploited, we realized ATLAS as a Telegram chat-bot, exploiting the open API provided by Telegram. The same travel assistant might be furnished via a different front-end, too.

Hereafter, we show how ATLAS runs in the Telegram chat-bot interface. The *chain of incremental refinements* that is dynamically set up from the execution of the following scenarios, is similar to that given in Fig. 9.

Local journey organization use case. Sara lives in Trento, Italy, and she wants to find her way to reach the Christmas markets located in Piazza Fiera. Her departure place is in via Fogazzaro. In Fig. 12, we show the relevant screenshots of the ATLAS chat-bot running on her smartphone.

Sara enters her departure and destination points (see the screenshot on the left side in Fig. 12). Being both places in the same city, Trento, a local planning would be more appropriate. Thus, the *Viaggia Trento* journey planner is dynamically selected. The journey planner's response is further handled and parsed to be showed on the chat-bot. The result is shown to Sara as in the central screenshot in Fig. 12. Since she opted for a healthy solution, the Viaggia Trento journey planner replies with a bike-sharing service, whose racks are close to both her source and destination places. At this point, to know if there are available bikes to be used, the travel assistant continues with its execution and it identifies the bike-sharing service available in Trento, namely *e-motion*[15]. It selects its fragment whose execution allows the application to get information about the available bikes at the closest bike-sharing racks. The result is shown as in the right-side screenshot in Fig. 12.

[13] https://api.tfl.gov.uk/
[14] https://www.citybik.es/
[15] http://www.provincia.tn.it/bikesharing

Three bikes up to 11 are still available at the rack close to Sara (first element in the result list). The e-motion bike-sharing service does not allow for the booking of bikes, so that the execution of ATLAS stops here.

Global journey organization use case. Paolo must organize his working journey from Trento to Torino. The relevant screenshots for his journey are reported in Fig. 13.

In this case, the travel assistant opts for a global planning solution served by the *Rome2Rio* global journey planner. The found travel alternatives are shown to Paolo as in the central screenshot in Fig. 13. Different alternatives are available (e.g., rideshare, bus, train, etc). Paolo selects the rideshare solution, which is also the less expensive. It is provided by the *BlaBlaCar* ride-sharing service. Further details about the selected solution are shown to Paolo, as in the right-side screenshot in Fig. 13. We highlight here that, to continue with the booking of the ride-share solution, it is required that he is subscribed to the BlaBlaCar service.

These execution examples exhibit two important aspects of our approach. Firstly, they show its *bottom-up* nature, where mobility services functionalities go through the domain objects hierarchy (refer to Fig. 3) till the user process where they are executed. Secondly, this happens in a completely transparent way for the user that interacts with only one application.

6.2.1 ATLAS evaluation

To evaluate ATLAS, both in terms of *effectiveness* and *efficiency*, we have run a set of experiments. The tests are done on real-world problem that were generated by randomly choosing an origin and a destination points. The specification of ATLAS used to evaluate it contains 14 domain object models, 17 fragment models and 12 types of domain properties. We ran ATLAS using a dual-core CPU running at 2.7 GHz, with 8 Gb memory. To show its *feasibility*, we evaluate the following aspects: (i) how long it takes to wrap up real services as domain objects; (ii) how much automatic refinement (service selection and composition) affects the execution of the travel assistant.

To answer to the first point, and based on our experience acquired during the development of ATLAS, we can argue the following. To wrap a real service as a domain object, the developer needs (i) to master the domain objects modeling notation and (ii) to understand the service behavior, its functionalities, its input/output data format and how to query it. Wrapping time clearly changes between experienced and non-expert developers. From our analysis, it ranges from 4 to 6 hours, considering average complex services. Moreover, it is also relevant to claim that this activity is done *una tantum*: after its wrapping, the service is seamlessly part of the approach and exploited for automatic composition and refinement.

To answer to the second point, we collected both the adaptation and mobility services execution statistics, to understand how long they take, on average, to be executed. To evaluate the automatic refinement, we carried

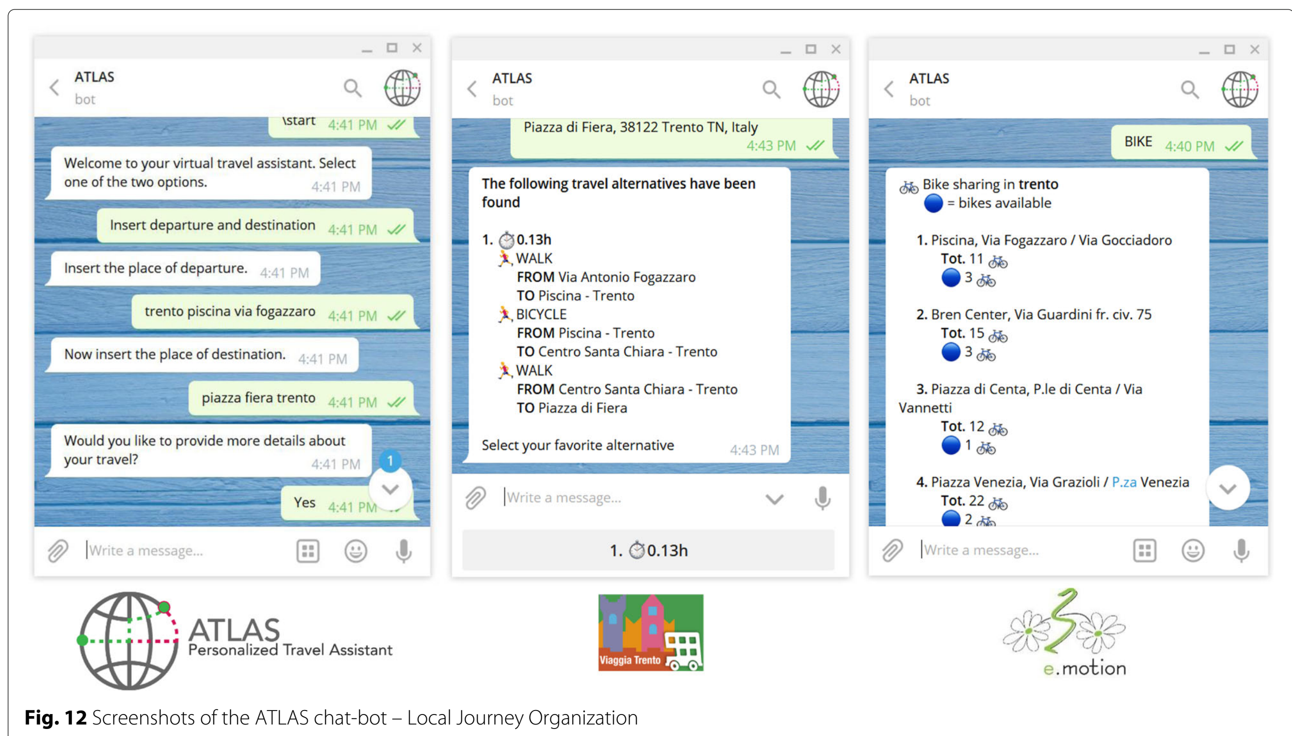

Fig. 12 Screenshots of the ATLAS chat-bot – Local Journey Organization

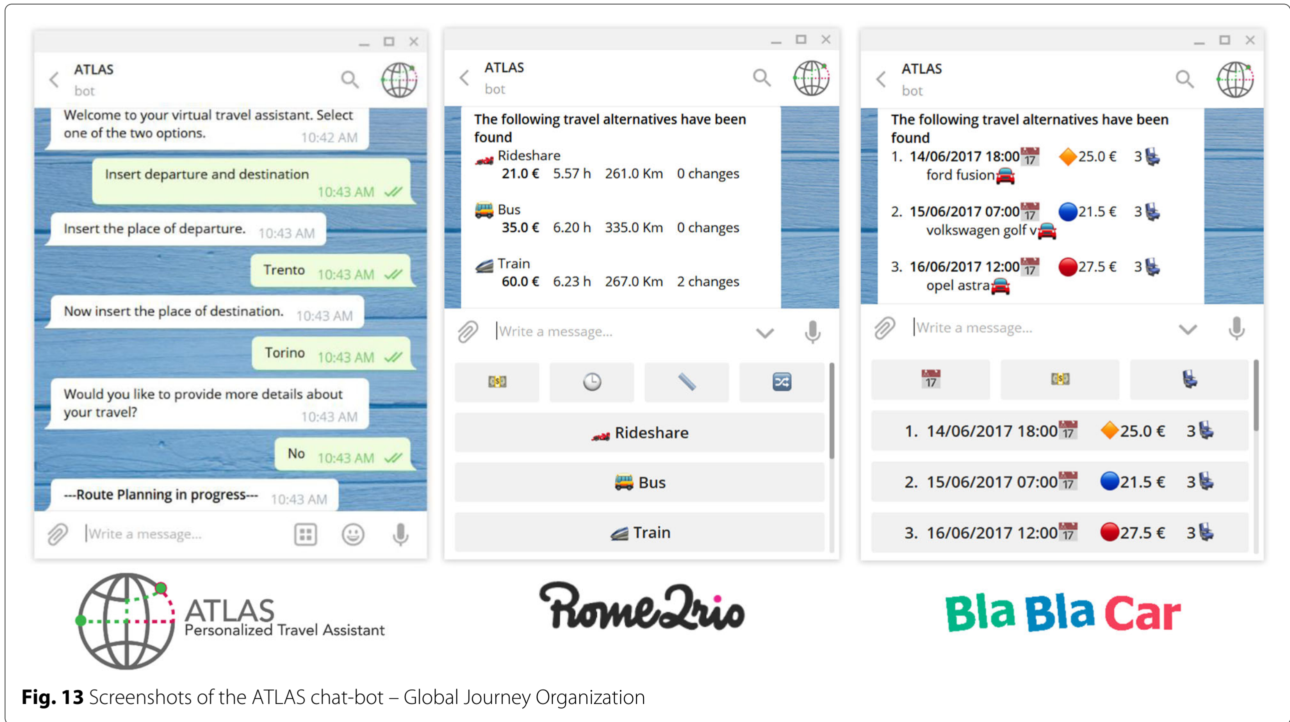

Fig. 13 Screenshots of the ATLAS chat-bot – Global Journey Organization

out an experiment in which we considered 10 runs of ATLAS handling various end-users' requests. We collected adaptation data such as the number of adaptation cases, their complexity and the time required to generate adaptation solutions. For each run, more than 150 refinement cases were generated. Figure 14 shows the distribution of problem complexity considering the 10 runs.

The complexity of an adaptation problem is calculated as the *total amount of transitions* in the state transition systems representations of the domain properties and fragments present in the problem. For simplicity, in the graph we aggregated the problem complexities in ranges of 20. The majority of the problems have a complexity in-between 0 and 19 transitions and 40 and 59 transitions. Notice that the occurrence of complex problems (complexity ranging from 80 to 100 transitions) is relatively rare (in this real-world battery of tests). Figure 15 shows the percentage of refinement problems solved within a certain time. We can see that, for all the runs, 93% of problems are solved within 0.2 s. Only 3% of the problems require more than 0.5 s to be solved, and the worst case is anyhow below 1.5 s.

To measure how much automatic refinement influences the execution of ATLAS, we compared the data about the time required for adaptation with the response time of real-world services wrapped in ATLAS. Figure 16 relates the (average) time required to solve a composition problem to the problem complexity.

The average time is computed considering in the 10 runs all the refinement problems having the same complexity. As expected, problems with higher number of transitions (and hence the most complex planning domain) take more planning time than problem with less complexity. Figure 17, instead, relates to the (average) response time of (a subset of) real mobility services, which are part of ATLAS.

We can notice that, in the worst case, the adaptation requires a time close to 1.5 seconds, while the services

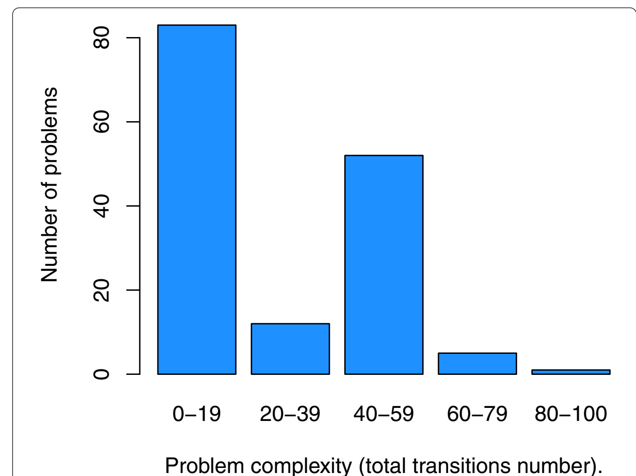

Fig. 14 Distribution of Problems Complexity

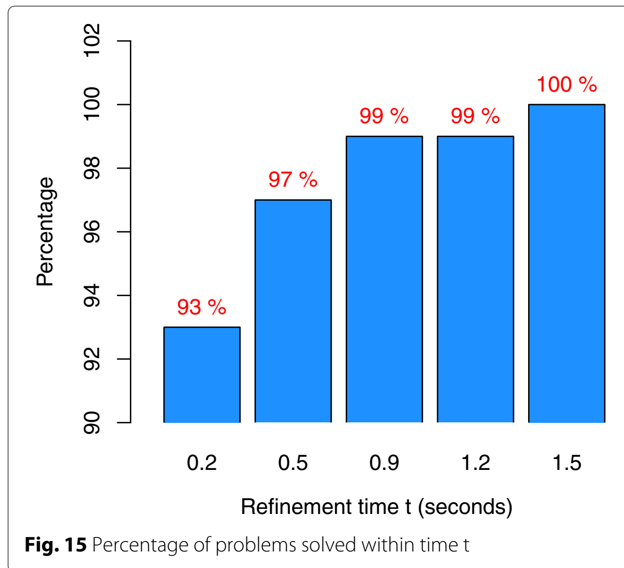

Fig. 15 Percentage of problems solved within time t

Services (Avg) Response Time	
Service Name	**Resp. Time**
Bla Bla Car	0.78 secs
City Bikes	0.23 secs
Google Transit	0.65 secs
Rome 2Rio	1.20 secs
Travel for London	3.20 secs
Viaggia Trento	0.77 secs

Fig. 17 (Average) Services Execution Time

response time ranges from 0.23 to 3.20 seconds. Moreover, the adaptation takes more time for the most complex problems that, however, are the less frequent to be executed. We can argue that the automatic refinement responsiveness is equivalent to that of mobility services.

Eventually, extended experimental results have been obtained in [16] where the presented approach has been used to realize an application in the IoT domain, where devices (e.g., sensors and actuators) act as service providers. To analyze the scalability of the approach, we measured the overall execution time of the application by considering up to 100 devices. Figure 18 shows the execution times (expressed in seconds) when varying the number of device instances. We found that the execution time values vary within a narrow interval, i.e., from 1.96 to 2.10 seconds.

In conclusion, these results demonstrate the *effectiveness* and the *efficiency* of our approach when applied to a real-world complex scenario.

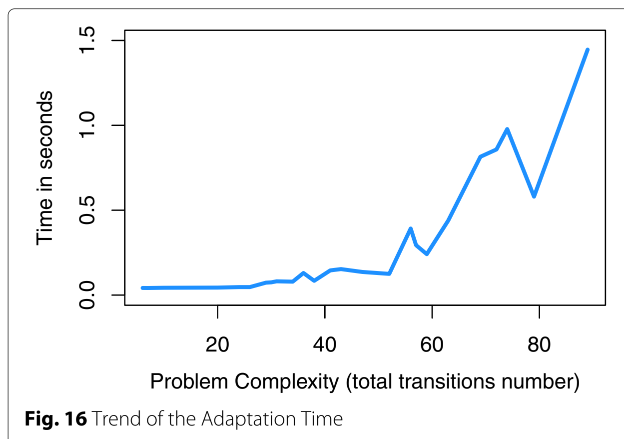

Fig. 16 Trend of the Adaptation Time

7 Lifecycle of the design for adaptation of service-based applications

In this section, we illustrate the *overall lifecycle* that we envisage for modeling and executing adaptive service-based applications, as depicted in Fig. 19. It gives a complete overview of the different perspectives of the approach (i.e., *modelling*, *adaptation*, *interaction*), the potentially involved actors (i.e., *platform provider*, *service providers*, *end-users*) and an abstraction of the main activities and artifacts.

In the following, each subsection is devoted to a particular perspective of the overall lifecycle (i.e., a row in Fig. 19). For each perspective, we also highlight the view of the different involved actors (i.e., a column in Fig. 19).

7.1 The modeling perspective

From a modeling perspective, each actor has a different view on the models of the application and is differently involved in its development and/or operation.

Platform provider view. The *Platform Provider*, with his team, is in charge to realize, maintain and provide to third parties a comprehensive platform allowing them to build and execute adaptive service-based applications on top of it. The design of the foundations for realizing adaptive service-based applications is made by the two models we have introduced in Section 4, namely the *Domain model* and the *Domain Objects model*. Given a specific domain (e.g., mobility), the *domain model* is specified by

	Exec. Time
# instances = 5	1.96 secs
# instances = 50	2.01 secs
# instances = 100	2.10 secs

Fig. 18 Scalability of the approach

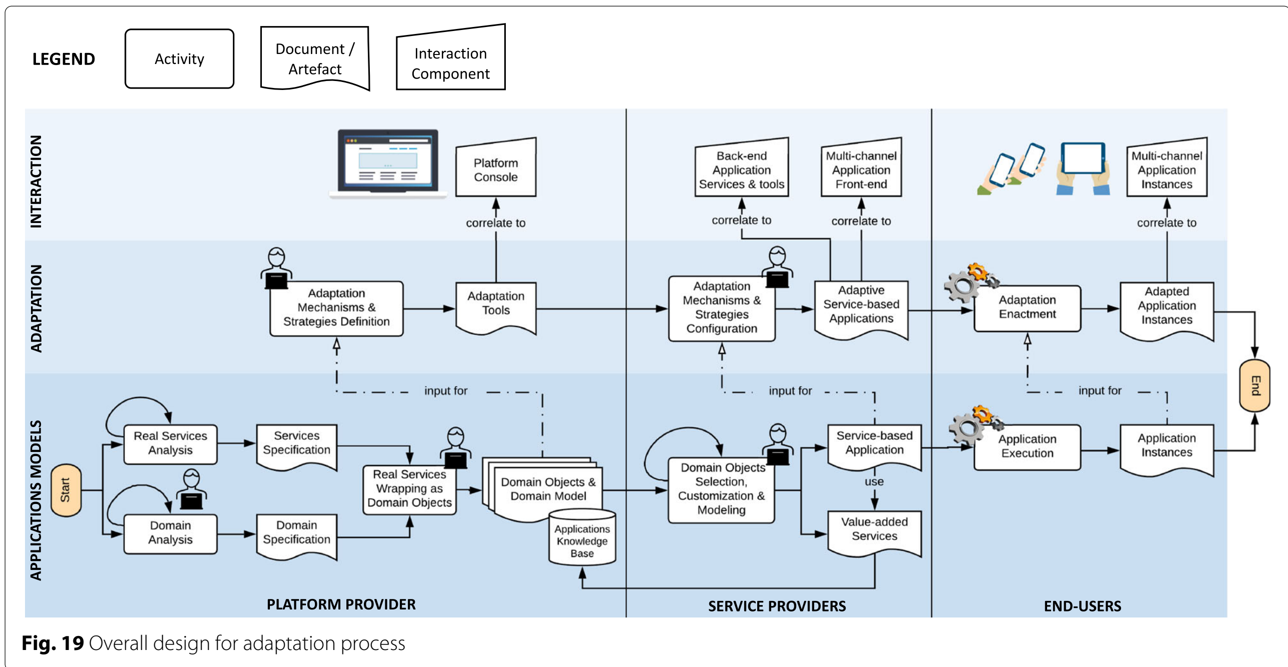

Fig. 19 Overall design for adaptation process

domain experts and it describes the operational environment of the application (see Domain Analysis activity and Domain Specification document in Fig. 19).

Furthermore, another important contribution from domain experts is an accurate analysis of the available services that are part of the targeted domain (see Real Service Analysis activity in Fig. 19). As outcome of the analysis, the domain experts release a high level description of the features, behavior, usage and offered functionalities of the analyzed services (see Services Specification document in Fig. 19).

The domain model and the services analysis constitute the input for the activity of defining the *domain object model*, accomplished by the application's developers (see Real Services Wrapping as Domain Objects activity in Fig. 19). In other words, developers wrap-up the services identified by domain experts as the *concrete implementations* of the *abstract concepts* in the domain model. This allows us also to overcome the typical mismatch among services' interfaces. The domain model and the domain objects models contribute to enrich the *application knowledge base* where they are stored (see Domain Objects and Domain Model artifact in Fig. 19).

We highlight that the activity of wrapping services as domain objects (or defining new ones) is not executed only once, and certainly not only during the initial design of an application. To the contrary, it is a continuous running activity, due to the continuous discovery and availability of new services (see the loop arrows on the Real Services Analysis and Domain Analysis activities in Fig. 19). Moreover, this activity can be performed as a collective co-development process [28], in a crowd-sourcing style [29], where each developer contributes to add new interesting services, thus enriching the application knowledge base. For these reasons we say that our approach supports the *continuous development* of service-based adaptive applications.

Service providers view. The role played by service providers is that of using and exploiting the tools, the engines and the models provided by the platform, in order to define, develop and execute their own service-based applications on top of it, such as ATLAS. This can be done by *selecting* and *customizing* the already available domain objects (see Domain Objects Selection, Customization & Modeling activity in Fig. 19) and by *defining new value-added services as domain objects* (see Value-added Services artifact in Fig. 19), together with the corresponding new domain concepts they implement. Also the domain objects defined by service providers can be stored in the application knowledge base and made available to the outside. This way they contribute to the continuous development of adaptive by design services and corresponding applications. Moreover, service providers can decide to develop and release their newly defined applications (see Service-based Application artifact in Fig. 19), by using whatever technologies.

End-users view. End-users are the final beneficiaries of the deployed service-based applications. Different application instances will be instantiated for different users (see Application Instances artifact in Fig. 19) and each instance will be characterized by its own *network of domain objects instances*, dynamically raised from the execution of

adaptable process. The domain objects network is made by instances of the domain objects corresponding to the services effectively exploited by the user.

7.2 The adaptation perspective

In this section we describe the adaptation perspective in the lifecycle of our approach, as depicted in Fig. 19–adaptation layer.

Platform provider view. The platform provider must supply all the tools and enablers (see Adaptation Tools artifact in Fig. 19) allowing the platform users (i.e., service providers) to define adaptive applications on top of it, as well as applications to effectively perform the adaptation, when executed. In other words, the *adaptation mechanisms and strategies* used by the platform (see Adaptation Mechanisms and Strategies Definition activity in Fig. 19) must be exposed in such a way that external users can benefit from and exploit them.

The platform provider must also provide to the platform's users a way to use the available adaptation tools, allowing them to understand and exploit the adaptation techniques, when defining their applications on top of the platform. In conclusion, other adaptation approaches can be exploited, as an alternative or in addition to the AI-based planning approach.

Service providers view. Different service providers can exploit the platform for defining their own applications or new value-added services. They have just to configure the adaptation mechanisms provided by the platform (see Adaptation Mechanisms & Strategies Configuration activity in Fig. 19). As a result, service providers will be able to release adaptive service-based applications (see Adaptive Service-based Applications artefact in Fig. 19) that can be customized and executed on top on the platform.

End-users view. The end-users use the available adaptive applications. They effectively enact the adaptation techniques (see Adaptation Enactment activity in Fig. 19). Indeed, adapted application instances (see Fig. 19) are dynamically created, customized and run over their requirements, based on their applications usage. This happens thanks to the adaptation mechanisms (e.g., local and refinement). Once the specific user execution environment is known, appropriate services can be selected, composed and exploited to satisfy the different user's goals.

7.3 The interaction perspective

From the operation and usage perspective of the application, each actor differently interacts with it (see the interaction level in Fig. 19).

Platform provider view. The platform provider, together with his team, is in charge of realizing the platform and its enablers, and then using it to realize and provide different adaptive service-based applications or simply adaptive services. In order to allow external service providers to exploit these applications through the platform, the platform provider should make available all the tools, the modeling environment and languages, the access to the different engines running in the platform, through an access console (see the Platform Console component in Fig. 19).

Service providers view. The service providers play a double role. From one side, they act as platform users. Indeed, they use the platform (i.e., its tools, enablers, engines, services) as a third-party service, or a PaaS (i.e., a platform as a service). To this aim, service providers access to and interact with the platform (see the Back-end Applications Services & Tools component in Fig. 19). From another side, service providers can decide to release their value-added services as applications. To this aim and from an interaction point of view, they can decide about the technologies to use for developing their applications (e.g., mobile apps, web applications) and also define the corresponding user interfaces (see the Multi-channel Application Front-end component in Fig. 19). While for the back-end of their applications service providers exploit the platform, for the front-end they are independent from the platform and its console.

End-users view. The end-users, finally, are not aware of the platform itself and exploit it in a completely transparent way. End-users just interact with the available applications through their interfaces, also using different devices, such as their smartphones, laptop, tablet and so on, depending on the specific technologies through which the service providers released their applications (see the Multi-channel Application Instances component in Fig. 19).

8 Related work

Many modern software systems are increasingly required to offer *continuous services* [30, 31]. Traditional software maintenance supports software evolution by providing updates that are applied *off-line*: the system is shut down, updated, and restarted. This solution, however, is not applicable when the system management must be carried out at runtime.

This need has motivated two parallel and independent approaches. Software engineers have started conceiving self-adaptive software systems [32, 33] that is, systems able to exploit internal capabilities to diagnose problems or changes in the context, and react accordingly. The advent of virtualized computing resources has also fostered DevOps [34] principles that suggest the idea of continuous evolution and release through the strict collaboration between development and operations. However, both solutions intrinsically embed some weaknesses. Conceiving a purely self-adaptive system means that *any* possible problem or change should be foreseen beforehand;

otherwise the system would not be able to react. While self-adaptation may be extremely effective at solving specific problems, widening their scope can be problematic. The analysis required to foresee potential issues may be expensive, and not always feasible. In contrast, focusing on rapidly changing implementations and on their automated deployment imposes *continuous changes*, even when they are not required, and can have severe consequences on the quality of released software.

Self-adaptation refers to the ability of a system to autonomously adapt at runtime, based on adaptation models, to maintain its non-functional requirements, by reacting to changes in the context it operates in [32, 33]. However, the increased interdependencies between software components and the complexity of execution contexts make the task of fully defining a priori adaptation need and solutions more difficult.

A variety of runtime adaptation approaches have been proposed in the literature. Within Dynamic Software Product Lines (DSPL) [11], the notion of software families, used to refer to common and reusable software assets [35, 36] is combined with predefined feature models that specify alternative variations that can be used for adaptation. These solutions have also been combined with aspect-oriented modelling methods for expressing self-adaptive systems at design time by separating implementation concerns [37, 38] and enable both design and runtime adaptations to meet new requirements [38]. Rule-based approaches have also been proposed [39–41] for defining adaptations. Cutting points in software models are identified at design time and rules are used to capture adaptations in terms of actions to take at different cutting points.

Context-oriented programming [42] has also been suggested as a paradigm for programming adaptable systems. In this case, adaptation relies on a pool of code variants chosen according to predefined program's context. Many languages have been extended (e.g., Lisp, Python, Ruby, Java) to integrate the notion of code fragments (e.g., methods or functions) that can be specialized with respect to each possible context. Adaptations have also been defined as (rule-based) mechanisms that allow the system to pass from one implementation logic, expressed in terms of behavior models, to another [39]. Similar solutions have been proposed for Mode Automata [43], Featured Transition Systems [44], and Labelled Transition System [45, 46] models. More recently, Artificial Intelligence planning frameworks have also been proposed [26, 47] in combination with state transition models of system behaviors, to address software adaptation in terms of a classical planning problem. In our approach, we also make use of AI planning to realize automated service compositions. Similarly, in [48] the authors automatically realize choreography-based service-oriented systems, by exploiting the CHOReVOLUTION approach [49]. In particular, the work proposed in [48] represents the concrete implementation of a real case study in the mobility domain employing the CHOReVOLUTION synthesis process that allows for realizing dynamic choreographies via distributed coordination of services. In the context of this work, we can not but argue about *microservices* and the *microservices architectural style* [8]. Most existing work on microservices focus on general architectural principles and migration guidelines [10, 50, 51]. Very rare works propose self-adaptation solutions in this context; Sampaio et al. [52] propose and approach to optimize microservice-based applications at runtime.

All the above mentioned approaches for adaptation rely on the underlying assumption of close world context and system. Consequently, adaptations can be pre-defined (e.g., rules-based models for adaptations are developed at design time) and any dynamic change in the software system components and/or functionality would require developers' intervention. Most approaches also provide methods where the implementation and adaptation logic are not clearly separated. This makes the systems' design much more complex and the runtime adaptation execution less flexible in managing dynamic context changes.

Our approach aims at providing the following contributions in the area of self-adaptive software systems:

- **Definition of models and programming paradigms for the development of software systems that are adaptable "by design"**, whereby runtime adaptations is not just an exception handling mechanism but an intrinsic characteristic of the system.
- **Development of adaptation mechanisms and strategies** for identifying the best runtime adaptations, without modifying the implementation logic to make software systems resilient to changes whilst preserving their qualities.

9 Critical discussion

We hereby discuss a set of limitations.

Data-driven composition requirements. Currently, composition requirements are expressed in terms of goals on abstract activities. In particular, they reflect functional properties of services allowing the definition of control-flow composition requirements. As a future work, we plan to define an extension of the domain properties such that to consider *data variables* related to context states.

Monitoring of unexpected events. It may happen that the execution of service-based applications is affected by unexpected events coming from the context that should be handled. These events are not devised by domain experts at design time, they are triggered by the operational context. In the current version of our approach, we

do not deal with the *monitoring of context events*. This limitation can be overcome by extending our approach with existing approaches [20] dealing with the monitoring of evolving contexts.

Further adaptation mechanisms and strategies implementation. We highlight that while the *Adaptation Engine* implements all the adaptation mechanisms and strategies reported in Section 5.1, in the implementation of our design for adaptation approach we currently handle the refinement mechanism. As future work we plan to extend the approach to the management of the other mechanisms and strategies. However, overcoming this limitation requires more for an implementation effort that for an extension of the approach, which already includes all the required constructs to handle the local and compensation adaptation mechanisms (e.g., preconditions, effects, goals).

Users involvement and flexible adaptations. When executing ATLAS, we can notice that the selection of the proper services is transparent to the user. From one hand, the service selection is indirectly affected by the user preferences. Obviously, it can also happen that there are no solutions satisfying the user's preferences, or the provided service composition represents an undesired outcome for the user. However, this strictly relates to the availability of services in the application, too. On the other hand, the indirect involvement of users in the service selection and composition tasks can be seen as a limitation of the approach. Nevertheless, it can be overcome in different ways. For instance, by considering all the different adaptation solutions that might satisfy a specific adaptation need, if more than one solution are available, and involve the user in the selection of the preferred one.

QoS-driven service selection and composition. Although the fragments discovery and selection is currently functional, the presented approach has been recently extended to include a QoS-driven service selection and composition [53]. Promising results have been obtained in the IoT scenario of [16].

9.1 Threats to validity

A threat to the *internal validity* of our approach is represented by the Process Engine, which currently operates in a centralized manner. Obviously, and this is part of our future work directions, it should evolve to better deal with the execution of applications running in distributed environments.

A threat to the *external validity* is that the presented results have been obtained on a set of case studies modeled by us, with the support of a developers group, comprising also experts in both the mobility and IoT domains. To increase the representativeness of the input models (i.e., domain model and domain object model) to our approach, further domain experts and software architects should be involved in a wider experimentation.

A second threat to the external validity consists in the fact that each service (or thing) needs to be wrapped-up as domain object for being available in the application. However, we plan to extend the approach to support the automated wrapping of services/things as domain objects, thus also enabling the definition of new goal types at runtime.

10 Conclusion and future work

The design for adaptation approach presented in this article is a proposal to solve the current open issues related to the modeling and execution of adaptive service-based applications. Its aim is to provide a complete solution for services management and exploitation, while considering the evolving nature of the environments in which they operate. Thanks to this general approach, we can facilitate services integration and interoperability, via service-based adaptive applications, thus better exploiting their functionalities and meeting users needs. By applying our approach, a novel ecosystem of customizable services that are easily personalized in different contexts can be designed, deployed, adapted and made available to the interested stakeholders. Indeed, it offers a lightweight-model, with respect to the existing languages for service modeling and adaptation, and it can be implemented with every object-oriented languages.

As already anticipated, different tasks represent our future work agenda, that are: (i) studying the usability of our approach by exposing the defined models and tools to users with different levels of experience; (ii) experimenting the approach on real applications coming from industrial experiences; and (iii) introducing automation in the initial activity of the design for adaptation process, by devising a technique to wrap-up services/things into domain objects.

Abbreviations

IoS: Internet of services; IoT: Internet of things; IoP: Internet of people; APFL: Adaptive pervasive flow language; STS: State transition systems; AI: Artificial intelligence; HOAA: Higher order abstract activity; ATLAS: A world-wide personAlized TraveL AssiStant; DSPL: Dynamic software product lines

Acknowledgments

Not applicable.

Authors' contributions

This manuscript is a contribution that originates from the doctoral thesis of M.D.S. All the authors contributed to the definition of the design for adaptation model for adaptive service-based applications. M.D.S. and A.B. carried out the implementation of the framework and of the travel assistant described in the case study. M.D.S. and A.B. wrote the manuscript with input from all authors. All authors read and approved the final manuscript.

Funding

Not applicable.

Availability of data and materials

The implementation of the ATLAS travel assistant framework is available at https://github.com/das-fbk/ATLAS-Personalized-Travel-Assistant. A

supporting video illustrating the main features and its live demonstration can be found at: https://vimeo.com/357367106.

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Gran Sasso Science Institute, Computer Science department, Viale Francesco Crispi, 67100 L'Aquila, Italy. [2]Fondazione Bruno Kessler, Via Sommarive, 18, 38123 Trento, Italy.

Received: 15 October 2019 Accepted: 13 March 2020
Published online: 27 March 2020

References

1. Pallis G. Cloud computing: The new frontier of internet computing. IEEE Internet Comput. 2010;14(5):70–73.
2. Moreno-Vozmediano R, Montero RS, Llorente IM. Key challenges in cloud computing: Enabling the future internet of services. IEEE Internet Comput. 2013;17(4):18–25.
3. Commission E. Next Generation Internet initiative. 2016. https://ec.europa.eu/digital-single-market/en/policies/next-generation-internet. Accessed 19 Mar 2020.
4. Group C-ETPX-E. Future Internet Strategic Research Agenda, Ver. 1.1. 2010. https://ec.europa.eu/programmes/horizon2020/en/h2020-section/future-internet. Accessed 19 Mar 2020.
5. Bouguettaya A, Singh MP, Huhns MN, Sheng QZ, Dong H, Yu Q, Neiat AG, Mistry S, Benatallah B, Medjahed B, Ouzzani M, Casati F, Liu X, Wang H, Georgakopoulos D, Chen L, Nepal S, Malik Z, Erradi A, Wang Y, Blake MB, Dustdar S, Leymann F, Papazoglou MP. A service computing manifesto: the next 10 years. Commun ACM. 2017;60(4):64–72.
6. Baresi L, Nitto ED, Ghezzi C. Toward open-world software: Issue and challenges. IEEE Comput. 2006;39(10):36–43.
7. Issarny V, Georgantas N, Hachem S, Zarras AV, Vassiliadis P, Autili M, Gerosa MA, Hamida AB. Service-oriented middleware for the future internet: state of the art and research directions. J Internet Serv Appl. 2011;2(1):23–45.
8. Lewis J, Fowler M. Microservices in a Nutshell. 2014. https://www.thoughtworks.com/insights/blog/microservices-nutshell. Accessed 19 Mar 2020.
9. Newman S. Building Microservices – Designing Fine-Grained Systems: O'Reilly Media; 2015. Accessed 19 Mar 2020.
10. Taibi D, Lenarduzzi V, Pahl C. Continuous architecting with microservices and devops: A systematic mapping study. In: Cloud Computing and Services Science - 8th International Conference, CLOSER 2018, Revised Selected Papers. Springer; 2018. p. 126–51.
11. Hinchey M, Park S, Schmid K. Building dynamic software product lines. IEEE Comput. 2012;45(10):22–26.
12. Bucchiarone A, De Sanctis M, Marconi A, Pistore M, Traverso P. Design for adaptation of distributed service-based systems. In: Service-Oriented Computing - 13th International Conference, ICSOC 2015, Proceedings; 2015. p. 383–93. https://doi.org/10.1007/978-3-662-48616-0_27.
13. Bucchiarone A, De Sanctis M, Marconi A, Pistore M, Traverso P. Incremental composition for adaptive by-design service based systems. In: IEEE International Conference on Web Services, ICWS 2016; 2016. p. 236–43. https://doi.org/10.1109/icws.2016.38.
14. Marchau V, Walker W, van Duin R. An adaptive approach to implementing innovative urban transport solutions. Transp Policy. 2008;15(6):405–12.
15. Bucchiarone A, Cappiello C, Nitto ED, Kazhamiakin R, Mazza V, Pistore M. Design for adaptation of service-based applications: Main issues and requirements. In: Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops - International Workshops, ICSOC/ServiceWave 2009, Revised Selected Papers; 2009. p. 467–76. https://doi.org/10.1007/978-3-642-16132-2_44.
16. Alkhabbas F, De Sanctis M, Spalazzese R, Bucchiarone A, Davidsson P, Marconi A. Enacting emergent configurations in the iot through domain objects. In: Service-Oriented Computing - 16th International Conference, ICSOC, Proceedings; 2018. p. 279–94. https://doi.org/10.1007/978-3-030-03596-9_19.
17. Eberle H, Unger T, Leymann F. Process fragments. In: On the Move to Meaningful Internet Systems: OTM 2009, Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009, Vilamoura, Portugal, November 1-6, 2009, Proceedings, Part I. Springer; 2009. p. 398–405.
18. Sirbu A, Marconi A, Pistore M, Eberle H, Leymann F, Unger T. Dynamic composition of pervasive process fragments. In: IEEE International Conference on Web Services, ICWS 2011, Washington, DC, USA, July 4-9, 2011; 2011. p. 73–80. https://doi.org/10.1109/icws.2011.70.
19. Bucchiarone A, Lluch-Lafuente A, Marconi A, Pistore M. A formalisation of adaptable pervasive flows. In: Web Services and Formal Methods, 6th International Workshop, WS-FM, Revised Selected Papers; 2009. p. 61–75. https://doi.org/10.1007/978-3-642-14458-5_4.
20. Saralaya S, D'Souza R. A review of monitoring techniques for service based applications. In: 2nd International Conference on Advanced Computing, Networking and Security, Mangalore, India, December 15-17; 2013. p. 96–101. https://doi.org/10.1109/adcons.2013.18.
21. Guermah H, Fissaa T, Hafiddi H, Nassar M, Kriouile A. Context modeling and reasoning for building context aware services. In: ACS International Conference on Computer Systems and Applications, AICCSA 2013; 2013. p. 1–7. https://doi.org/10.1109/aiccsa.2013.6616439.
22. Bucchiarone A, Marconi A, Pistore M, Raik H. A context-aware framework for dynamic composition of process fragments in the internet of services. J Internet Serv Appl. 2017;8(1):6–1623.
23. Bertoli P, Pistore M, Traverso P. Automated composition of web services via planning in asynchronous domains. Artif Intell. 2010;174(3-4):316–61.
24. Raik H, Bucchiarone A, Khurshid N, Marconi A, Pistore M. Astro-captevo: Dynamic context-aware adaptation for service-based systems. In: Eighth IEEE World Congress on Services, SERVICES 2012, Honolulu, HI, USA, June 24-29, 2012; 2012. p. 385–92. https://doi.org/10.1109/services.2012.14.
25. Marconi A, Pistore M, Traverso P. Automated composition of web services: the ASTRO approach. IEEE Data Eng Bull. 2008;31(3):23–26.
26. Bucchiarone A, Marconi A, Mezzina CA, Pistore M, Raik H. On-the-fly adaptation of dynamic service-based systems: Incrementality, reduction and reuse. In: Service-Oriented Computing - 11th International Conference, ICSOC, Proceedings; 2013. p. 146–61. https://doi.org/10.1007/978-3-642-45005-1_11.
27. Bucchiarone A, De Sanctis M, Marconi A. ATLAS: A world-wide travel assistant exploiting service-based adaptive technologies. In: Service-Oriented Computing - 15th International Conference, ICSOC 2017, Proceedings; 2017. p. 561–70. https://doi.org/10.1007/978-3-319-69035-3_41.
28. Deck M, Strom M. Model of co-development emerges. Res-Technol Manag. 2002;45(3):47–53.
29. Estellés-Arolas E, González-Ladrón-de-Guevara F. Towards an integrated crowdsourcing definition. J Inf Sci. 2012;38(2):189–200.
30. Shahin M, Babar MA, Zhu L. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. IEEE Access. 2017;5:3909–43.
31. Chen L. Microservices: Architecting for continuous delivery and devops. In: IEEE International Conference on Software Architecture, ICSA 2018; 2018. p. 39–46. https://doi.org/10.1109/icsa.2018.00013.
32. In: Cheng BHC, de Lemos R, Giese H, Inverardi P, Magee J, editors. Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]. Lecture Notes in Computer Science, vol. 5525: Springer; 2009.
33. de Lemos R, Giese H, Müller HA, Shaw M, (eds). Software Engineering for Self-Adaptive Systems, 24.10. - 29.10.2010. Dagstuhl Seminar Proceedings, vol. 10431. Germany: Schloss Dagstuhl - Leibniz-Zentrum für Informatik; 2010.
34. Bass L, Weber I, Zhu L. DevOps: A Software Architect's Perspective: Addison-Wesley; 2015.
35. Cubo J, Gámez N, Fuentes L, Pimentel E. Composition and self-adaptation of service-based systems with feature models. In: Safe and Secure Software Reuse - 13th International Conference on Software Reuse, ICSR 2013, Proceedings; 2013. p. 326–42. https://doi.org/10.1007/978-3-642-38977-1_25.

36. Murguzur A, Trujillo S, Truong HL, Dustdar S, Ortiz Ó,., Sagardui G. Run-time variability for context-aware smart workflows. IEEE Softw. 2015;32(3):52–60.
37. Popovici A, Alonso G, Gross TR. Just-in-time aspects: efficient dynamic weaving for java. In: AOSD; 2003. https://doi.org/10.1145/643603.643614.
38. Parra C, Romero D, Mosser S, Rouvoy R, Duchien L, Seinturier L. Using constraint-based optimization and variability to support continuous self-adaptation. In: Proceedings of the ACM Symposium on Applied Computing, SAC 2012; 2012. p. 486–91. https://doi.org/10.1145/2245276.2245370.
39. Ehrig H, Ermel C, Runge O, Bucchiarone A, Pelliccione P. Formal analysis and verification of self-healing systems. In: Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Proceedings; 2010. p. 139–53. https://doi.org/10.1007/978-3-642-12029-9_10.
40. Yu J, Sheng QZ, Swee JKY. Model-driven development of adaptive service-based systems with aspects and rules. In: Web Information Systems Engineering - WISE 2010 - 11th International Conference, Hong Kong, China, December 12-14, 2010. Proceedings; 2010. p. 548–63. https://doi.org/10.1007/978-3-642-17616-6_48.
41. Hussein M, Han J, Yu J, Colman A. Enabling runtime evolution of context-aware adaptive services. In: 2013 IEEE International Conference on Services Computing; 2013. p. 248–55. https://doi.org/10.1109/scc.2013.77.
42. Hirschfeld R, Costanza P, Nierstrasz O. Context-oriented programming. J Object Technol. 2008;7(3):125–51.
43. Maraninchi F, Rémond Y. Mode-automata: About modes and states for reactive systems. In: Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Proceedings; 1998. p. 185–99. https://doi.org/10.1007/bfb0053571.
44. Cordy M, Classen A, Heymans P, Legay A, Schobbens P. Model checking adaptive software with featured transition systems. In: Assurances for Self-Adaptive Systems - Principles, Models, and Techniques; 2013. p. 1–29. https://doi.org/10.1007/978-3-642-36249-1_1.
45. Schaefer I, Poetzsch-Heffter A. Using abstraction in modular verification of synchronous adaptive systems. In: Workshop "Trustworthy Software" 2006, May 18-19, 2006, Saarland University, Saarbrücken, Germany. Germany: Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl; 2006.
46. Zhang J, Cheng BHC. Using temporal logic to specify adaptive program semantics. J Syst Softw. 2006;79(10):1361–9.
47. Marrella A. Automated planning for business process management. J Data Semant. 2019;8(2):79–98.
48. Autili M, Salle AD, Gallo F, Pompilio C, Tivoli M. A choreography-based and collaborative road mobility system for l'aquila city. Futur Internet. 2019;11(6):132.
49. Autili M, Salle AD, Gallo F, Pompilio C, Tivoli M. Chorevolution: Automating the realization of highly-collaborative distributed applications. In: Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Proceedings; 2019. p. 92–108. https://doi.org/10.1007/978-3-030-22397-7_6.
50. Bucchiarone A, Dragoni N, Dustdar S, Larsen ST, Mazzara M. From monolithic to microservices: An experience report from the banking domain. IEEE Softw. 2018;35(3):50–55.
51. Francesco PD, Lago P, Malavolta I. Architecting with microservices: A systematic mapping study. Journal of Systems and Software. 2019;150: 77–97.
52. Sampaio AR, Rubin J, Beschastnikh I, Rosa NS. Improving microservice-based applications with runtime placement adaptation. J Internet Serv Appl. 2019;10(1):4–1430.
53. De Sanctis M, Spalazzese R, Trubiani C. Qos-based formation of software architectures in the internet of things. In: Software Architecture - 13th European Conference, ECSA 2019, Proceedings; 2019. p. 178–94.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com