Domain Objects for Continuous Context-Aware Adaptation of Service-Based Systems

Conference Paper · June 2013

DOI: 10.1109/ICWS.2013.82

6 authors, including:

Antonio Bucchiarone
Fondazione Bruno Kessler
86 PUBLICATIONS **954** CITATIONS
SEE PROFILE

Annapaola Marconi
Fondazione Bruno Kessler
62 PUBLICATIONS **903** CITATIONS
SEE PROFILE

Paolo Traverso
Fondazione Bruno Kessler
180 PUBLICATIONS **8,976** CITATIONS
SEE PROFILE

Raman Kazhamiakin
Fondazione Bruno Kessler
56 PUBLICATIONS **955** CITATIONS
SEE PROFILE

Domain Objects for Continuous Context-Aware Adaptation of Service-based Systems

Antonio Bucchiarone, Annapaola Marconi,
Marco Pistore, and Paolo Traverso
Fondazione Bruno Kessler, Via Sommarive, 18, Trento, Italy
{*bucchiarone,marconi,pistore,traverso*}*@fbk.eu*

Piergiorgio Bertoli and Raman Kazhamiakin
SAYservice, Trento, Italy
{*bertoli,raman*}*@sayservice.it*

***Abstract*—The idea to build systems based on services, by reusing and combining software made available independently via different technologies and channels and dynamically organizing them into coherent processes, has a very well recognized potential. Achieving this potential stands crucially on the ability to recognize and exploit the context in which such applications operate, in terms of the available services, of the actual setup at run-time, and of the interaction with human actors. Changing contexts require run-time reaction, by adapting the ongoing overall process enacted by the service-based system to unexpected deviations. Continuous context-aware adaptation is hencefore a strong requirement in this setting. In this paper, we propose a service delivery platform designed for the continuous context-aware adaptation of service-based systems, based on the idea of monitoring the underlying context via a hierarchical model, and using the context to drive the choice and execution of adaptive pervasive flows. We describe the approach through an example from the smart cities e-mobility domain.**

***Keywords*-Service-Based Systems, Context-Awareness, Adaptation, Monitoring**

I. Introduction

Service-based systems , which are constructed by composing and configuring available services, have the significant advantage to reduce costs and efforts in the development, deployment, and maintenance of complex distributed applications. However, this potential advantage is hard to achieve in practice, due to the highly dynamic environments in which service-based systems must often operate. Indeed, the *context* in which service-based systems must operate continuously changes, starting from the change of the situation in which services are executed, to the availability of services, the availability of providers, the human actors interacting with the application as well as their requirements and preferences. Moreover, most often, the only way the application can react to such changing environment is at run-time, since different situations, with the corresponding available services and providers, are not known a priori. *Continuous context-aware adaptation* becomes therefore the key enabling property for service-based systems to cope with the dynamics of the continuously changing environment in which they must operate.

In this paper, we propose a service delivery platform specifically designed for the continuous context-aware adaptation of service-based systems. The platform is based on the idea of using *domain objects* as a way to connect the operational environment of the system with the system context, meant as a high level representation of such environment, which can be monitored to trigger adaptation. Domain objects will be organized in a hierarchical network whose higher levels represent application-related concepts, whereas lower levels are closely connected to services and to the interactions with users; as such, they will are able to raise the abstraction level at which the domain evolution is perceived by the execution of services to the system context that allows the platform to monitor the changes in the environment. The system context is monitored by the execution of *adaptive pervasive flows*, an extension of traditional work-flows that annotate work-flow activities with preconditions, i.e. properties that must be satisfied by the current context, and whose failure is the trigger to run-time adaptations.

Based on this framework, we define and implement a set of different adaptation mechanisms based on automated planning techniques. They allow to deal with the case in which services are not known or available a priori. In this case, adaptive pervasive flows contain goals to be achieved, rather than links to executable services. Automated planning is performed at run-time and provides the ability to automatically refine goals into proper compositions of executable services by taking into account the whole current context. Automated planning at run time is also used to support the adaptation to unexpected context changes. Context changes that lead to the violation of preconditions trigger the planning for bringing the system to a situation where the process execution can be safely resumed. The planning techniques for different kinds of adaptation can then be combined in different adaptation strategies that can be used in different situations.

We describe the proposed approach through a running example taken from a realistic smart city e-mobility application, where the service-based system needs to deal with the variability of the services and actors involved, as well as of the situations in which it must operate. The application needs to react and to adapt to different user requirements and preferences at run-time, to the change of status of mobility services (parking garages, buses availability, car-pooling and car-sharing), to the required payment procedures to use them, and to all the real-time context changes that may affect user's mobility (e.g., train cancellation, bus delays, traffic jams).

The rest of the paper is structured as follows. We first describe the smart city mobility scenario. We then describe the general framework and approach, discussing the conceptual architecture of the framework, which is based on domain objects and adaptive pervasive flows which allow the monitoring of the environment and the triggering of adaptation. We then describe in detail the adaptation mechanisms, how they can be combined, and how they are realized by advanced automated planning techniques. In the final section we describe some related work and draw some conclusions.

II. Motivating Scenario: Smart Mobility Services

Supporting citizens mobility within the urban environment is a priority for municipalities worldwide. Although a network of multi-modal transport systems (e.g., buses, trains, metro), services (e.g., car sharing, bike sharing, car pooling, car rental), and smart technologies (e.g., sensors for parking availability, smart traffic lights, integrated transport pass) are necessary to better manage mobility, they are not sufficient. Citizens must be offered accurate travel information, where and when such information is needed to take decisions that will make their journeys more efficient and enjoyable.

In order to deliver "smart services" to citizens, in a smart territory the available systems should not be considered each by itself, but tightly interconnected in a synergic manner constituting a system of systems. Our goal is to develop an application, the Smart Mobility System, that, exploiting in a synergistic manner the heterogeneous city transport facilities, provides accurate, real-time, and customized mobility services supporting the whole travel duration.

The system needs to deal with the *dynamicity* of the scenario, both in terms of the variability of the actors and systems involved, and of the context changes affecting its operation. The application should support *customization*, which means that each mobility solution should take into account the user profile (e.g., travel impairments, preferred transport means), the user context (e.g., work commute, leisure trip with the family), the mobility facilities and services that can be used in the specific context (e.g., parking garages may be closed or may be full, parking payment services may require a registration). The application should also be *open*, which means that new transport facilities, such as a new car-pooling service, as well as changes in existing facilities (e.g., an RFID based bus pass) should be easily integrated in the application. Finally, the application needs to support not only the *planning phase of the journey*, but also promptly detect *real-time context changes* that may affect the journey (e.g., train cancellation, bus delays, traffic jams), and possibly *propose alternative solutions* to the user.

III. General Framework and Approach

In this section we present our approach to provide Smart Mobility System applications that are able to match all the requirements described in the previous section.

The proposed approach enables the continuous monitoring and adaptation of service-based systems, and is based on the one side on the ability to monitor systems to extract information on the context, and on the other on using such context to continuously adapt the ongoing process by appropriately composing reusable process fragments. We first discuss our approach at a conceptual level, before entering more in detail in the modeling of the key elements of the framework, that is the context and the process fragments.

A. Approach

Our overall framework is designed around a minimal set of elements, as depicted in Figure 1. It realizes a reactive loop where: (a) the entities that define the domain, both humans and systems (e.g., transportation, traffic, parking, and so on) are monitored, (b) the overall status of the system is understood, and problems with respect to the ongoing processes are identified, so that (c) the execution of these processes can be suitably adapted to the new status.

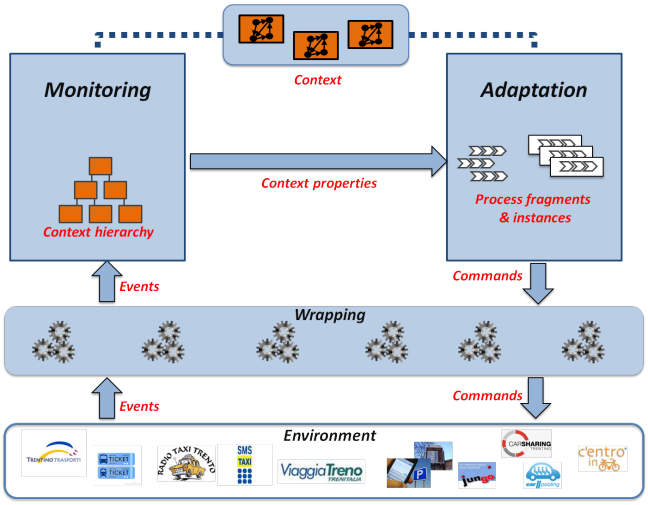

Figure 1: The overall approach

In this loop, the starting point is the possibility to interact with all the entities that contribute to the overall process. In the case of *systems*, the interaction exploits all the IT interfaces these systems expose on the web. These IT interfaces are however fragmented and heterogeneous, i.e., they consist of software designed independently from each other, without an holistic view of an overall system, and are made available through a large variety of different technologies (web pages, web APIs, REST or SOAP services, feeds, open data, and so on). There is hence a need to encapsulate interactions with systems, so that they are presented to the remaining components in a standardized way. In our approach, this is achieved by a specific "wrapping" layer, which has the goal to cope with the encapsulation of fragmented, heterogeneous, loosely structured and non-reliable sources, aiming at presenting them as open, uniform and reliable *services*. Each service exposes a set of *events*, carrying information and representing changes in the corresponding system, and a set of *commands*, i.e., of operations that can be performed on these systems. Technologically, this requires a set of capabilities, covering extraction of structured information out of loosely-structured sources, mashup (i.e., aggregation, filtering, transformation) of this information, data validation and caching, detection and diagnosis of problems in the behavior of the sources.

All the information made available by existing systems and exposed as events, is captured by a *monitoring* component, which is in charge to aggregate different pieces of information, and to convert it into a coherent description of the status of the system. For this purpose, the monitoring component must comprise abstraction, correlation and integration mechanisms that bring the technical pieces of information produced by the wrapping level to a level of representation which is appropriate to be managed at the business level.

The outcome of the monitoring component is the *context*, which is meant to model the status of the operational environment of the system. In our approach, the context consists of a structure of hierarchically connected elements,

named *domain objects*, each of which is meant to represent a concept of the domain at hand, at some level of abstraction; while the higher levels of the network represent process-related concepts, the lower levels are closely connected to the kind of information produced by systems.

The context is the input to the *adaptation* component, which is in charge of analyzing it for the purpose of checking the progress of the ongoing processes, and, whenever needed, for adapting them to the changes in the environment. The adaptation component stands on top of the definition of reusable and adaptable *process fragments*. The adaptation component is also in charge of executing the (possibly adapted) process, producing the commands that are issued back to the wrapping layer, for delivery to the systems and users.

In the following, we discuss in more detail the modeling of such two key concepts at the basis of our architecture, namely the processes and the context.

B. Modeling

1) Process Modeling.: The system operation is modeled through a set of *entities* (e.g., public transport providers, parking facilities, car and bike sharing providers, end-users), each specifying its behavior through a *business process*. Unlike traditional system specifications, where the business processes are static descriptions of the expected run-time operation, our approach allows to define dynamic business processes that are refined at run time according to the features offered by the system. This kind of modeling extremely simplifies the specification of business processes, since the developer does not need to think about and specify all the possible alternatives to deal with specific situations (e.g., context changes, availability of functionalities, improbable events).

The underlying idea is that entities can join the system dynamically, publish their functionalities through a set of *process fragments* that can be used by other entities to interoperate, discover fragments offered by the other entities, and use them to automatically refine their own business processes. For instance, within the Smart Mobility System, if the *user* needs assistance to reach the city center by car, it discovers the fragments provided by the *municipality* and by the different *parking facilities*. These fragments model the specific procedures for searching for the most convenient/available parking, to reach the selected parking, to pay for the stay. Different fragments may be provided by different parking facilities according to their type (e.g., on-street park meter parking, private garage) and to the services they offer (e.g., real-time info on availability, smart-card/SMS payment procedure). Moreover, if the user chose a peripheral parking, he will discover also all those fragments that can be used to reach the final destination (e.g., public transports, taxi, walking direction fragments)

To model business processes and fragments, we use $CAptLang$ [5], an extension of traditional workflow languages (e.g., BPEL) which makes them suitable for adaptation and execution in dynamic pervasive environments. In addition to the classical workflow language constructs (e.g., input, output, data manipulation activities, complex control flow constructs), $CAptLang$ add the possibility to relate the process execution to the system context by annotating activities with *preconditions* and *effects*. *Preconditions* constrain the activity execution to specific context configurations, and in our framework are used to catch violations in the expected behavior and trigger run-time adaptation. *Effects* model the expected impact of the activity execution on the system context, and are used to automatically reason on the consequences of fragment/process execution. Consider for instance the *Private Parking* fragment of Figure 2. A precondition of the activity *ReachParking* is that the chosen parking facility is available. The same activity can be annotated with the effect *p.reached* to model the fact that the expected impact of this activity is to make the make the system context evolve to a configuration where the chosen parking has been reached. Similarly, a precondition of the *Confirm Payment* activity in the *SMS Payment* fragment is that the payment procedure was successful.

Moreover, the $CAptLang$ allows to specify *abstract activities* within fragments. An abstract activity is defined at design time in terms of the *goal* it needs to achieve, expressed as context configurations to be reached, and is automatically refined at run time into an executable process, considering the set of available fragments, the current context configuration, and the goal to be reached. For instance, the abstract activity *Park* of the parking process model in Figure 2, aiming at parking the car, is annotated with the goal *G1: p.status=reached and p.payment=started*. At run time, a specific fragment composition will be generated to achieve this goal taking into account the preferences of the user, the status of the parking facilities, and the available fragments for reaching the selected parking and correctly starting the payment procedure (e.g., park meter pay for on-street parkings). Activities can also be annotated with a *compensation goal* that has to be fulfilled any time adaptation requires to rollback the process instance and they have already been successfully executed.

Formally, we model process fragments as state transition systems, where each transition corresponds to a particular fragment activity. In particular, we distinguish four kinds of activities: input and output activities model communications among processes; concrete activities model internal elaborations by the process; and abstract activities correspond to the abstract activities of the process.

Definition 1: (Process Fragment) Process Fragment defined over context property diagrams O is a tuple $p = \langle S, S_0, A, T, \mathit{Ann} \rangle$, 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;
- $\mathit{Ann} = \langle \mathit{Pre}, \mathit{Eff}, \mathit{Goal}, \mathit{Comp} \rangle$ is a process annotation, where $\mathit{Pre} : A_{in} \cup A_{out} \cup A_{con} \rightarrow L^*$ is the precondition labeling function, $\mathit{Eff} : A_{in} \cup A_{out} \cup A_{con} \rightarrow E^*_{cnt}$ is the effect labeling function, $\mathit{Goal} : A_{abs} \rightarrow L^*$ is the goal labeling function, and $\mathit{Comp} : A \rightarrow L^*$ is the compensation labeling function.

We denote with $S(p)$, $A(p)$, etc. the corresponding elements

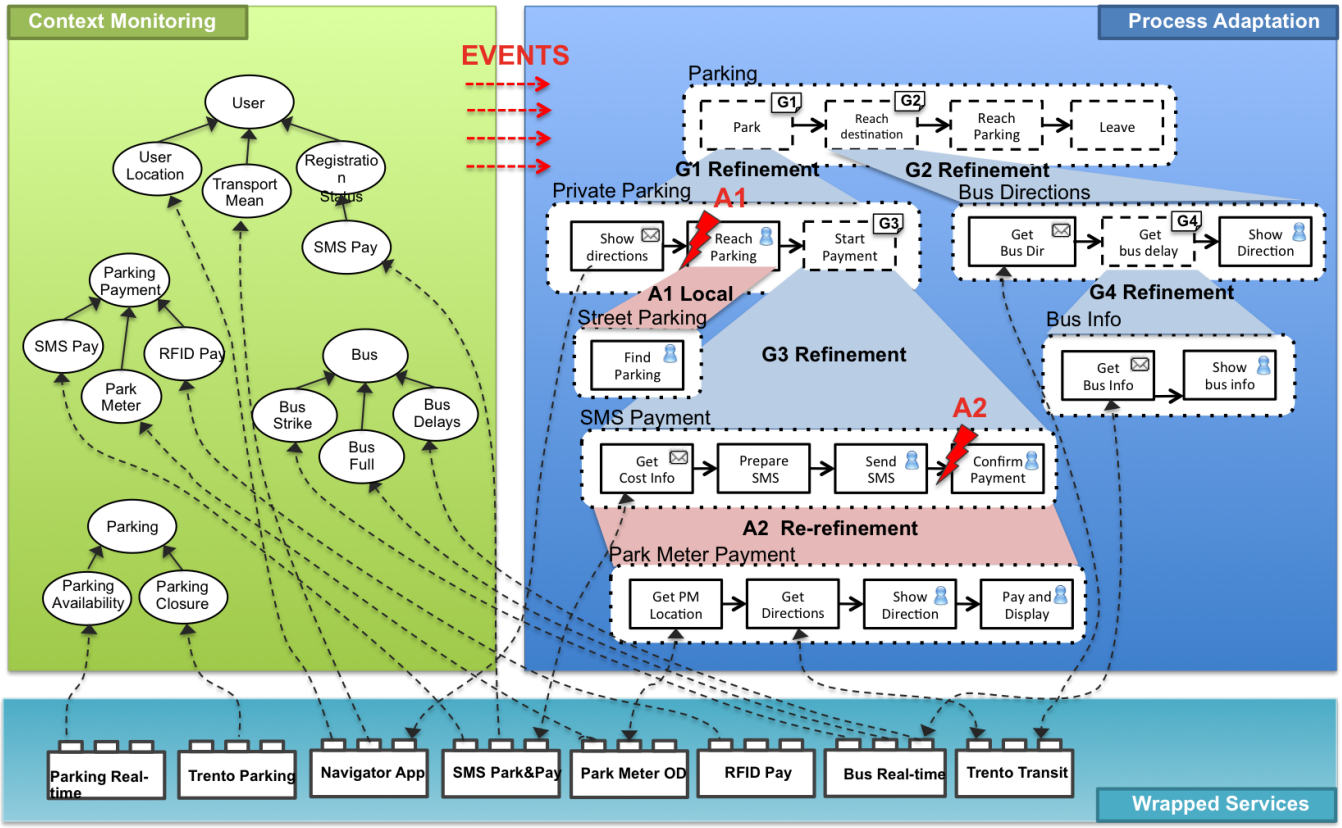

Figure 2: Adaptation and monitoring mechanisms in the Smart Mobility scenario

of a fragment p.

2) Context Modeling and Monitoring.: The dynamic features offered by the framework rely on the existence of a *context model* which describes the operational environment of the system. The context is defined through a set of *context properties*, each describing a particular aspect of the system domain (e.g., current location of a user, bus delays, availability of parking places). A context property may evolve as an effect of the execution of a fragment activity, which corresponds to the normal behavior of the domain (e.g., the car has been parked in the chosen parking, the parking payment is successful), but also as a result of exogenous changes (e.g., bus delay, parking area full, parking payment procedure failed) . A *context configuration* is a snapshot of the context at a specific time, capturing the current status of all its context properties.

In terms of modeling, we start from a minimalistic view and model a context property as a *domain object*, by which we mean a state-transition system capturing all possible property values and value changes. Each transition is labeled with a corresponding *event*.

Definition 2: (Domain Object) A Domain Object is a tuple $o = \langle L, L_0, E, T \rangle$, where:

- L is a set of context states and $L_0 \subseteq L$ is a set of initial states;
- E is a set of context events;
- $T \subseteq L \times E \times L$ is a transition relation.

We denote with $L(o)$, $E(o)$, etc. the corresponding elements of domain object o. In general, the overall context comprises a set of domain objects O, and hence, the context state results from the combination (i.e., the cartesian product) of the states of its domain objects. Formally a space of *context configurations* is defined as $L = \prod_{o \in O} L(o)$.

This notion needs to be extended in order to allow the monitoring component to connect the technical wrapping of systems and users with the context-based view provided in terms of domain objects, hence raising the abstraction level at which the domain evolution is perceived, from the technical level — at which the wrapping is executed and technical events are captured — to the business level, which is presented in terms of domain objects that evolve on the basis of business-level events.

To achieve this, the monitoring component considers an extended notion of context, where domain objects are linked to each other to form a hierarchically structured conceptual network, where objects at different levels in the hierarchy provide different abstractions; the objects at top level build in fact the context, while the lower-level objects are closely related to the wrapping layer. In the hierarchy, a layer of objects "monitors" the layers below; this of course implies that an extended notion of domain object is devised, where not only the object evolves on the basis of some event, but also, it may produce some proprietary events that are amenable to monitoring by higher-layer objects. As an example, consider the *Parking Payment* domain object in Figure 2. The status of the payment depends on the specific payment procedure adopted (e.g., payment via SMS, via park meter, via smart card) and thus the domain object updates this information

according to the events triggered by the lower-layer objects (i.e., *SMS Pay*, *Park Meter*, *RFID Pay*). This is captured by the following extended definition, where the transition relation also encompasses the ability of the object to fire a set of proprietary events.

Definition 3: (Active Domain Objects) An Active Domain Object is a tuple $o = \langle L, L_0, E, E_{OUT}, T \rangle$, where:

- L is a set of context states and $L_0 \subseteq L$ is a set of initial states;
- E is a set of events (the "input" events to the object);
- E_{OUT} is a set of events (the "output" events from the object), such that $E \cap E_{OUT} = \emptyset$;
- $T \subseteq L \times E \times L \times 2^{E_{OUT}}$ is a transition relation.

Again, we denote with $L(o)$, $E(o)$, etc. the corresponding elements of the active domain object o.

On the basis of this, the notion of hierarchical context used in the monitoring component simply consists of directed acyclic graph of active domain objects that are connected in a proper way, i.e., where the events produced by an object are fully captured by all objects monitoring it.

Definition 4: (Hierarchical Context) A Hierarchical Context is a pair $h = \langle O, H \rangle$, where:

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

Given this definition, it is easy to express which are the input events for a hierarchical context, that is all the events that affect its evolution and are not generated by some object in the hierarchy; this set of course must contain all the events possibly generated by the wrapping layer.

IV. Adaptation Problem and Solution

The problem of adapting a service-based system on the basis of a dynamically changing context poses several conceptual challenges which and reflect into strict technical requirements for the concrete realization of the adaptation mechanisms. First, the adaptation must obey to a criteria of *incrementality*, implying that adaptation goals must be in general tackled by successive refinements, up to their full solution. In doing so, care must be taken in that each adaptation impacts only those portions of the system that need be affected. Second, a *continuous adaptation* process must be enforced, where adaptations can be applied also on adapted (portions of) systems, so to be able to revise formerly introduced adaptations, which may turn out to be inappropriate as the context may further change; this may as well include full backtracking on some portions of applied adaptations. Third, since the monitored services may provide partial information about the environment, and they may expose only partially its dynamics, the adaptation mechanisms must deal with *imperfect and incomplete information*; that is, they must be able to produce and enact meaningful and appropriate adaptations even if the context is not precisely and fully known. Remarkably, this comes together with the fact that the changes in the context are in most cases due to unforeseeable and uncontrollable decisions taken by some actors in the domain; hence, adaptations must be identified in the face of uncontrollability and uncertainty.

In the following subsections, we first introduce high-level mechanisms that satisfy these requirements, and then we describe their technical realization, which goes through (a) the modeling of the problem, (b) the linkage to the above-mentioned mechanisms, and finally (c) the resolution of the problem via the conversion into a form of planning problem for which techniques powerful enough exist to deal with the above requirements.

A. Adaptation Mechanisms

We now present different adaptation mechanisms that can be used to handle the dynamicity of context-aware pervasive systems. They have been already successfully applied for a car logistic scenario in [4], and offer an adequate starting point for the adoption in broader settings, such as the one at hand. In general, our framework can deal with two different adaptation needs: the need for refining an abstract activity within a process instance, and the need to cope with a violation of the context precondition of an activity that has to be executed.

The *refinement mechanism* is triggered whenever an abstract activity in a process instance needs to be refined. The aim of this mechanism is to automatically compose available process fragments taking into account the goal associated to the abstract activity and the current context configuration. The result of the refinement is an executable process that composes a set of available fragments and, if executed, fulfills the goal of the abstract activity. The advantage of performing refinements at run-time is twofold: available fragments are not always known at design time (e.g., a new parking payment procedure may be activated and the corresponding fragment added to the system), and the composition strongly depends on the current execution context (e.g., a parking area may be full or closed and thus its fragments not usable, a bus service may not be active in the late evening). Consider, for instance, the abstract activity *Park* of the main parking car process in Figure 2. During the execution, based on the parking availability and user preferences, the activity is automatically refined with the *Private Parking* fragment. The obtained refinement is injected in the parking process instance that can continue its execution and achieve its goal.

Composed fragments may also contain abstract activities which requires further refinements during the process execution. This is the case of the *Start Payment* activity in the *Private Parking* refinement. Since the user is registered to the *SMS Park&Pay* service and the selected parking supports this type of payment, the activity is refined with the *SMS Payment* fragment that allows, given the current user location, to obtain the relevant information (e.g., parking identifier, cost information) to automatically prepare the SMS for starting the payment. The result of this incremental refinement is a multi-layer process execution model (see Figure 2), where the top layer is the initial process of the entity and intermediate layers correspond to context-aware incremental refinements.

The *local adaptation mechanism* aims at identifying a solution that allows for re-starting the execution of a faulted

process from a specific activity. To achieve this, a composition of fragments is generated and its execution brings the system to a context configuration satisfying the activity precondition. As an example, consider adaptation *A1* of Figure 2. The parking process is executing the *Reach Parking* activity of the *Private Parking* fragment, however the chosen parking gets full and the precondition of the activity is not valid. The aim of local adaptation in this case is to find an alternative available parking location. The most convenient solution identified by the system is to search for a street parking nearby, through the *Street Parking* fragment. After executing the local adaptation process, the parking process instance can resume the execution of the original process.

The *compensation mechanism* can be used to dynamically compute a compensation process for a specific activity. The compensation process is a composition of fragments specifically selected for the current context and whose execution fulfills the compensation goal. The advantage of specifying activity compensation as a goal on the context, rather than explicitly declaring the activities to be executed (e.g., as in BPEL), is in the possibility to dynamically compute the compensation process taking into account the specific execution context (e.g., cancel a booking of a train ticket / parking lot in case of changes in the planned journey).

When different adaptation mechanisms are combined and executed in a precise order, *adaptation strategies* are realized. They are able to deal with complex adaptation needs that cannot be addressed by applying adaptation mechanism in isolation. An example is the case where a violation of an activity precondition cannot be resolved with local adaptation. For instance, in adaptation *A2* of Figure 2 there is no way of making the SMS parking payment successful if the prepaid user credit is used up and there are no available fragment to recharge it. Similarly, there is no way of solving a delay of a bus or of a train. Another example is the delay of an abstract activity refinement, caused by unavailability of fragments to be composed to fulfill the goal within a specific execution context.

Our framework provides different ways of combining adaptation mechanisms. A first possibility is *one shot adaptation*, where the different adaptation mechanisms are combined for a single adaptation problem, and a comprehensive solution is searched for and, if found, executed. Another possibility is *incremental adaptation*, where each adaptation mechanism in the strategy is called and the resulting adaptation process is executed before applying the next adaptation mechanism. This interleaving of adaptation and execution makes it possible to tailor each adaptation to the specific execution context, but has the main drawback of not knowing in advance whether the strategy can be completely executed.

In the following we present some adaptation strategies that we have identified and that resulted to be very useful in our scenario, whose patterns can all be implemented through one-shot or incremental adaptation.

The *re-refinement* strategy can be applied whenever a faulted activity belongs to the refinement of an abstract activity. The aim of this strategy is to compensate all the activities of the refinement labeled with a compensation goal and that have been already executed (through compensation mechanism) and to compute a new refinement (through refinement mechanism) that satisfies the goal of the abstract activity and taking into account the new context configuration. This strategy is used in the scenario of Figure 2 to solve the adaptation need *A2*, where the parking SMS payment is not successful. In this specific execution context, the re-refinement of the abstract activity *StartPayment* is re-refined with a fragment composition that, taking into account the parking supported payment facilities, allows the user to pay through a park meter (see *Park Meter Payment* refinement in Figure 2).

Another strategy is *backward adaptation* that aims at bringing back the process instance to a previous activity in the process that, given the new context configuration, may allow for different execution decisions. This strategy requires the compensation of all the activities that need to be rolled back (compensation mechanism), and for bringing the system to a configuration where the precondition of the activity to be executed is satisfied (local adaptation).

Other strategies can be defined by composing the adaptation mechanisms in a different way by combining sub-strategies, hence easily adding other patterns to the framework.

B. Modeling and solving adaptation problems

We now discuss how the adaptation problem can be solved; this goes through two key steps: a formal modeling of the problem, and the conversion of such problem into a well-know form of problem, that is a specific form of planning. Once this is achieved, we are in a position to leverage on existing and effective planning tools to implement the approach.

1) Adaptation problem formal statement: As illustrated in Figure 2, the process is a hierarchical structure, obtained through the refinement of abstract activities into fragments. Following this, a process configuration can be modeled as a list of triples process-activity-history: the first element in the list describes the fragment currently under execution, the current activity, and history of past activities executed in the current fragment; the other triples describe the hierarchy of ancestor fragments, each one with abstract activities currently under execution and past history. As we will see, the history of past activities is necessary whenever a backward adaptation is performed, to determine which compensations need to be executed.

Definition 5: (Process Configuration) We define a process configuration as a non-empty list of triples $E_p = (p_1, a_1, h_1), (p_2, a_2, h_2) \dots (p_n, a_n, h_n)$, where:

- p_i are process fragments;
- $a_i \in A(p_i)$ are activities in the corresponding process fragments, with $a_i \in A_{abs}(p_i)$ for $i \geq 2$ (i.e., all activities that are refined are abstract);
- $h_i \in A(p_i)^*$ is the sequence of past activities executed in the process fragment.

The configuration of the whole system is defined by the current configuration of the context properties, by the configuration of the processes in the system, and by the set of available fragments.

Definition 6: (System Configuration) Given a set O of context property diagrams, we define a system configuration for O as a tuple $\mathcal{S} = \langle \mathcal{I}, \mathcal{E}, \mathcal{F} \rangle$, where:

- $\mathcal{I} \in L(o_1) \times \ldots \times L(o_n), o_i \in O$ is the current configuration of context property diagrams;
- $\mathcal{E} \in E_{p_1} \times \ldots \times E_{p_n}$ is the current configuration of running processes;
- $\mathcal{F}$ is the set of available fragments.

We denote with $\mathcal{I}(\mathcal{S})$, $\mathcal{E}(\mathcal{S})$, etc. the corresponding elements of a system configuration $\mathcal{S}$.

For lack of space, we do not give a formal definition of evolution of a system configuration. Intuitively, the system evolves in three different ways. First, through the execution of, and interaction among, processes: this happens according to the standard rules of business process execution. Second, through the entrance (and exit) of new entities into the system: each new entity corresponds to the introduction of a new process in $\mathcal{E}$ and the instantiation of the corresponding context property diagrams; moreover, since entities can bring new fragments, it also corresponds to the extension of set $\mathcal{F}$. Third, through the execution of adaptations, which will be discussed in detail in the following.

We can now define a general concept of adaptation problem, which will be used to formalize the different adaptation mechanisms. Namely, a general adaptation problem contains complete information about the system configuration and an adaptation goal, which captures the adaptation objectives to be fulfilled.

Definition 7: (Adaptation Problem) Adaptation problem is a tuple $\xi = \langle \mathcal{S}, \mathcal{G} \rangle$, where $\mathcal{S}$ is the current system configuration and $\mathcal{G}$ is an adaptation goal over O.

We denote with $\mathcal{S}(\xi)$ and $\mathcal{G}(\xi)$ the corresponding elements of an adaptation problem ξ.

The solution to a general adaptation problem ξ is a process M_{adapt} that is obtained as the composition of a set of fragments in $\mathcal{F}(\mathcal{S})$. When executed from the current system configuration $\mathcal{S}$, and in the absence of exogenous events corresponding to unpredicted situations, M_{adapt} ensures that the resulting context configuration satisfies the goal $\mathcal{G}(\xi)$.

2) Modeling Adaptation Strategies as Adaptation Problems: We now show how adaptation strategies (and mechanisms), used to adapt a process instance E_p in a system configuration $\mathcal{S}$, can be transformed into general adaptation problems $\xi = \langle \mathcal{S}, \mathcal{G} \rangle$, and how the obtained process M_{adapt} is exploited to adapt system configuration $\mathcal{S}$ into a new configuration $\mathcal{S}'$. We assume that the adaptation is applied to fragment $E_p \in \mathcal{E}(\mathcal{S})$, with $E_p = (p_1, a_1, h_1), (p_2, a_2, h_2), \ldots (p_n, a_n, h_n)$, We also assume that a_0 is the first activity of M_{adapt}.

Refinement. The refinement mechanism is used whenever a_1 is an abstract activity that needs to be refined. This means to generate an adaptation solution for problem ξ where $\mathcal{G}(\xi) = \mathit{Goal}(a_1)$. $\mathcal{S}'$ is obtained from $\mathcal{S}$ by chaining the new fragment in the configuration of process p: $E'_p = (M_{adapt}, a_0, []) \cdot E_p$.

Local Adaptation. Local adaptation can be used to solve a situation where a_1 cannot be executed since $\mathit{Pre}(a_1)$ is violated. This means to generate an adaptation solution for problem ξ where $\mathcal{G}(\xi) = \mathit{Pre}(a_1)$. $\mathcal{S}'$ is obtained from $\mathcal{S}$ by updating E_p into $E'_p = (p'_1, a_0, h_1), (p_2, a_2, h_2), \ldots (p_n, a_n, h_n)$ as follows: p'_1 is obtained by merging M_{adapt} into p_1 before activity a_1.

Compensation. This mechanism has the objective to (partially or completely) rollback the past activities performed in fragment p_1, by applying the associated compensations. We assume that $h_1 = h_0, \tilde{a}_1, \ldots, \tilde{a}_m$, and that we want to rollback activities $\tilde{a}_1, \ldots, \tilde{a}_m$. This means to generate an adaptation solution for problem ξ where $\mathcal{G}(\xi) = \mathit{Comp}(\tilde{a}_m) \cdot \mathit{Comp}(\tilde{a}_{m-1}) \cdots \mathit{Comp}(\tilde{a}_1)$ is the concatenation, in reverse order, of the compensation goals associated to the activities that need to be rolled back. $\mathcal{S}'$ is obtained from $\mathcal{S}$ by updating E_p into $E'_p = (p'_1, a_0, h_0), (p_2, a_2, h_2), \ldots (p_n, a_n, h_n)$, where p'_1 is obtained by merging M_{adapt} into p_1 before activity $\tilde{a}_1$.

Adaptation Strategies. As described in Section **??**, adaptation mechanisms can be combined into complex adaptation strategies. The formalization of strategies into adaptation problems is straightforward: *incremental adaptation* is obtained by interleaving computation of adaptation problem and execution of the resulting adaptation process for each mechanism, while *one shot adaptation* of adaptation problems $\xi_1 = \langle S, \mathcal{G}_1 \rangle$ and $\xi_2 = \langle S, \mathcal{G}_2 \rangle$ is defined as $\xi_1 \cdot \xi_2 = \langle S, \mathcal{G}_1 \cdot \mathcal{G}_2 \rangle$.

3) Using planning to solve adaptation.: We now discuss how AI planning is exploited to implement the adaptation mechanisms and strategies discussed in Section IV-A. In particular, we need to elect planning techniques which are powerful enough to cope with uncertain and incomplete information, and to express goals associated to adaptation strategies. For these reasons, we adopt and adjust the service composition approach presented in [3]. According to it, a service composition problem is transformed into a planning problem and planning techniques are used to resolve it. Similarly, we transform an adaptation problem 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 [10].

A planning domain is derived from adaptive problem ξ. In particular, a set of n fragments $(p_1, \ldots, p_n \in \mathcal{F}(\mathcal{S}))$, m context property diagrams $(c_1, \ldots, c_m \in O)$ are transformed into state transition systems (STS) using transformation rules similar to those presented in [3]. While encoding fragments and context property diagrams as STSs, we remove all uncontrolled events, since they describe exogenous, improbable events. With these measures, the adaptation plan will be built under the assumptions that no exogenous events and service failures happen during the execution of the adaptation process.

The planning domain Σ is obtained as the product of the STSs $\Sigma_{p_1} \ldots \Sigma_{p_n}$ and $\Sigma_{c_1} \ldots \Sigma_{c_m}$, where STSs corresponding to processes are synchronized on inputs and outputs, and STSs for processes and for context properties are synchronized on preconditions and effects. $\Sigma = \Sigma_{p_1} \| \ldots \| \Sigma_{p_n} \| \Sigma_{c_1} \| \ldots \| \Sigma_{c_m}$ The initial state r of the planning domain is derived from the current configuration $\mathcal{I}(\mathcal{S})$, by interpreting it as states of the STSs defining the planning domain. Similarly, the adaptation goal $\mathcal{G}(\xi)$ is transformed into an EAGLE planning goal ρ by interpreting the configurations in $\mathcal{G}(\xi)$ as states in the planning domain.

Finally, we apply the approach of [11] to domain Σ and planning goal ρ and generate a plan Σ_c that guarantees achieving goal ρ once "executed" on system Σ. State transition system Σ_c is further translated into executable process M_{adapt}, which implements the identified adaptation strategy.

We remark that the planner may fail to find a suitable plan for a planning problem, if no such plan exists. In this case, the adaptation strategy needs to take care of this failure, e.g., according to the *incremental adaptation* approach.

V. Related Work and Conclusions

In this paper we have proposed an approach to monitor and adapt complex service-based systems, where the adaptation solutions are automatically derived at run time taking into account the actual state of the execution environment, obtained in terms of a context through a monitoring component that collects, aggregates and abstracts information appropriately, and the available fragments that can be used to solve the problem. We have defined different adaptation mechanisms (i.e., refinement, local, compensation), described how they can be combined through adaptation strategies to tackle complex adaptation problems, and shown how each adaptation problem can be transformed into an AI planning problem.

The problem of monitoring business processes and services is widely addressed in the literature. In particular, when applied to SOA compositions, monitoring refers to the problem of verifying functional and QoS properties of BPEL orchestrations [8], [2] or individual services [9], [14]. Several approaches consider explicitly also contextual properties of the former. These approaches address the "technical" aspects of executions (e.g., service quality, process performance) and do not explicitly address the issue of formally ensuring that the concurrent flows of information is correctly propagated.

Various approaches supporting adaptation have been defined, e.g., triggering repairing strategies as a consequence of a requirement violation [15], and optimizing QoS of service-based applications [18], [19], or for satisfying some application constraints [16]. Repairing strategies could be specified by means of policies to manage the dynamism of the execution environment [1], [6] or of the context of mobile service-based applications [13]. The aim of the strategies proposed by the aforementioned approaches range from service selection to rebinding and application reconfiguration [12], [17]. These are interesting features, but cannot deal with complex process restructuring and lack a design approach to support designers in developing applications where context-awareness and adaptivity are key embedded characteristics. An adaptation approach similar to ours is presented in [7]. SmartPM is a formal framework based on situation calculus and IndiGolog language that provides runtime adaptation of a process to deal with unplanned exceptions. The problem of adaptation is eventually reduced to a classical planning problem. However, SmartPM is currently a theoretical framework and the efficiency and overhead of adaptation modeling cannot be evaluated so far. Classical planning cannot deal with non-deterministic services, which are a natural property of modern SOA systems. As opposed to SmartPM, our approach is able to deal with stateful and non-deterministic services. Our formal framework intuitively represents the relevant business concepts (context, activity preconditions and effects, processes) and, in combination with tools translating adaptable pervasive flows to state transition systems, provides efficient development tools that minimize the adaptation modeling effort.

Our work lends to several extensions in different directions. First, our adaptation mechanism is currently applied to instances of business processes. We plan to extend our work to use the execution history of adapted instances as a training set to progressively improve the process model (*process evolution*) and the adaptation strategies to be used at process instance level. We are also investigating the possibility of using more sophisticated optimality criteria for planning adaptation activities, considering not only the length of the resulting plan but also minimal side effects, minimal execution cost of the process, and other quality indicators.

References

[1] L. Baresi, S. Guinea, and L. Pasquale. Self-healing BPEL processes with Dynamo and the JBoss rule engine. In *ESSPE'07*, pages 11–20. ACM, 2007.

[2] L. Baresi, S. Guinea, M. Pistore, and M. Trainotti. Dynamo + astro: An integrated approach for bpel monitoring. In *ICWS 2009*, pages 230–237, 2009.

[3] P. Bertoli, R. Kazhamiakin, M. Paolucci, M. Pistore, H. Raik, and M. Wagner. Control Flow Requirements for Automated Service Composition. In *Proc. ICWS'09*, pages 17–24, 2009.

[4] A. Bucchiarone, A. Marconi, M. Pistore, and H. Raik. Dynamic Adaptation of Fragment-based and Context-aware Business Processes . In *Proc. ICWS'12*, 2012.

[5] A. Bucchiarone, C. Antares Mezzina, and M. Pistore. Captlang: a language for context-aware and adaptable business processes. In *VaMoS'13*, pages 12:1–12:5. ACM, 2013.

[6] M. Colombo, E. Di Nitto, and M. Mauri. Scene: A service composition execution environment supporting dynamic changes disciplined through rules. In *ICSOC*, pages 191–202, 2006.

[7] M. de Leoni. Adaptive Process Management in Highly Dynamic and Pervasive Scenarios. In *Proc. YR-SOC*, pages 83–97, 2009.

[8] C. Ghezzi and S. Guinea. Run-Time Monitoring in Service-Oriented Architectures. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 237–264. Springer, 2007.

[9] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *J. Network Syst. Manage.*, 11(1), 2003.

[10] A. Marconi, M. Pistore, and P. Traverso. Implicit vs. explicit data-flow requirements in web service composition goals. In *ICSOC*, pages 459–464, 2006.

[11] A. Marconi, M. Pistore, and P. Traverso. Automated Composition of Web Services: the ASTRO Approach. *IEEE Data Eng. Bull.*, 31(3):23–26, 2008.

[12] H. Pfeffer, D. Linner, and S. Steglich. Dynamic adaptation of workflow based service compositions. In *ICIC '08*, pages 763–774. Springer-Verlag, 2008.

[13] E. Rukzio, S. Siorpaes, O. Falke, and H. Hussmann. Policy based adaptive services for mobile commerce. In *WMCS'05*. IEEE Computer Society, 2005.

[14] A. Sahai, V. Machiraju, M. Sayal, A. P. A. van Moorsel, and F. Casati. Automated SLA Monitoring for Web Services. In *DSOM 2002*, pages 28–41, 2002.

[15] G. Spanoudakis, A. Zisman, and A. Kozlenkov. A service discovery framework for service centric systems. In *IEEE SCC*, pages 251–259, 2005.

[16] K. Verma, K. Gomadam, A. P. Sheth, J. A. Miller, and Z. Wu. The METEOR-S approach for configuring and executing dynamic web processes. Technical report, University of Georgia, Athens, 2005.

[17] Y. Yan, P. Poizat, and L. Zhao. Self-adaptive service composition through graphplan repair. In *ICWS*, pages 624–627, 2010.

[18] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *WWW '03*, pages 411–421, 2003.

[19] Y. Zhai, J. Zhang, and K. Lin. Soa middleware support for service process reconfiguration with end-to-end qos constraints. In *ICWS*, pages 815–822, 2009.