2012 IEEE 19th International Conference on Web Services

Dynamic Adaptation of Fragment-based and Context-aware Business Processes

Antonio Bucchiarone, Annapaola Marconi, Marco Pistore and Heorhi Raik
Fondazione Bruno Kessler, Via Sommarive, 18, Trento TN 38123, Italy
{*bucchiarone,marconi,pistore,raik*}*@fbk.eu*

***Abstract*—We propose a comprehensive framework for adaptivity of service-based applications, which exploits the concept of process fragments as a way to model reusable process knowledge and to allow for the dynamic, incremental, context-aware composition of such fragments into adaptable service-based applications. The framework provides a set of adaptation mechanisms that, combined through adaptation strategies, are able to solve complex adaptation problems. An implementation of the proposed solution is presented and evaluated on a real-world scenario from the logistics domain.**

***Keywords*-Dynamic service adaptation, Dynamic service composition, Context-aware service-based applications**

I. Introduction

One of the key advantages of the service oriented paradigm is the possibility to reduce the development and maintenance cost of software applications without loosing the control of their quality and the capability of managing their lifecycle. A key enabling factor to fully exploit these advantages, is the capability of service oriented applications to adapt, i.e., to modify their behavior and to evolve in order to satisfy new requirements and to fit new situations. Addressing this problem is not at all easy, especially considering the challenges posed by the Internet of Services, where applications need to deal with a continuously changing environment, both in terms of the context in which the applications operate, and of the services, users and providers involved. In such a setting, the same application shall operate differently for different contextual situations, deal with the fact that involved services are not known a priori, and be able to dynamically react to changes to better fit the new situations.

Several approaches have been investigated in recent years to support the adaptation of service-based systems [1], [6], [10], [19], [22], [23], [4]. Unfortunately enough, despite the considerable effort dedicated to tackling this problem, we are still far from effective solutions. Most adaptation approaches require to analyze all the possible adaptation cases at design-time, and to embed the corresponding recovery activities in the system model [1], [8], [15]; they can hardly be used in dynamic settings, where adaptation cases are too many, and recovery activities cannot be defined at design time. Dynamic adaptation approaches, where adaptation activities are automatically derived at run-time on the basis of the current execution environment and of the specific adaptation need, try to solve these limitations. Most of these approaches [6], [10], [19], [22], [23], however, have a limited scope and deal with local forms of adaptation (e.g., service replacement). Moreover, the role of the execution environment is seldomly taken into account, thus preventing to capture sources of change that are external to the application, but still very relevant for its operation.

In this paper we propose a comprehensive framework for adaptivity of service-based application, which exploits the concept of process fragments [11] as a way to model reusable process knowledge and to allow for a dynamic, incremental, context-aware composition of such fragments into adaptable service-based applications. The framework allows for business processes that are only partially specified at design time, and that are automatically refined at run-time taking into account the specific execution context. This refinement exploits the available fragments, which are provided by the other actors and systems to describe the services and capabilities that are offered to the process in the specific context. The framework also support run-time adaptation to unexpected or improbable context changes that may affect the execution of the application. This is achieved through a set of adaptation mechanisms that, if properly combined through adaptation strategies, find solutions to bring the application to a state where the execution can be correctly resumed. The realization of the different adaptation mechanisms is achieved extending sophisticated AI planning techniques for the automated composition of services [17].

The proposed approach has been applied to a car logistic scenario based on the operation of the sea port of Bremen, for which a demonstrator has been implemented [**?**]. The scenario, dealing with the management of cars from their arrival by ship to their delivery to retailers, entails a high level of complexity both in terms of the number of actors involved (e.g., cars, ships, terminals, storage areas, treatment facilities, trucks), each having its own regulations and procedures, and in terms of the dynamicity of the execution environment (e.g., availability of port facilities, accidental damages, human errors, changes in regulations).

The rest of the paper is structured as follows. Section II introduces the Car Logistic scenario that is used as a reference throughout the paper; Section III presents the proposed framework for modeling adaptable context-aware service-based applications, the adaptation mechanisms and strategies for the dynamic process refinement and the handling of context changes; Section IV gives a formal specification of the framework and shows how AI planning techniques are used to solve adaptation problems; finally, Section V describes how we have evaluated our solution using the Car Logistic demonstrator while Section VI presents some related works, and conclusions.

II. Motivating Scenario

The scenario used throughout the paper is based on the operation of the sea port of Bremen, Germany [3], where nearly 2 million new vehicles are handled each year in order *to deliver them from manufacturers to retailers*. The delivery process of each car consists of a set of procedures

978-0-7695-4752-7/12 $26.00 © 2012 IEEE
DOI 10.1109/ICWS.2012.56

that can be customized according to the car brand, model, retailer-specific requirements, etc. Cars arrive by ship and are unloaded and unpacked at a certain terminal. Once a car is unpacked, it has to be moved to one of the storage areas, depending on the type of the car (e.g., covered/guarded areas for luxury cars) and on availability of parking spaces; different storage areas have different parking procedures that need to be followed. A car remains at the storage area until it is ordered by a retailer. Once a car stored is ordered, it continues its way towards the delivery. In particular, a car is treated at dedicated treatment areas (e. g., washing, painting, equipping, repairing) according to the details in the order. When a car is ready to be delivered it is moved to the assigned delivery gate, where it is loaded onto a truck, and eventually delivered to the retailer.

Our goal is to develop an application (the Car Logistic System (CLS)) to support the management and operation of the port, where numerous actors (i.e., cars, ships, trucks, treatment areas, etc.) need to cooperate in a synergistic manner respecting their own procedures and business policies. The system needs to deal with the dynamicity of the scenario, in terms of both the variability of the actors' procedures (customizable processes), and the exogenous context changes affecting its operation. Customization means that different brands and models of cars should be treated in a similar but customizable way. Moreover new car models, having specific requirements and procedures, have to be able to be easily integrated in the application. Similarly, the application needs to flexibly deal with changes in the procedures of external actors such as ships and trucks. Finally, the application needs to promptly reflect changes in international regulations and laws. Concerning context dynamicity, examples of environment conditions to be taken into account are unavailability or malfunctioning of the port facilities, accidental damages of cars and trucks, human errors (e.g., a car is parked in the wrong parking lot).

III. General Approach

A. Application Model

In this Section we present our framework for modeling adaptable context-aware fragment-based systems such as the CLS scenario described in Section II.

The system operation is modeled through a set of *entities* (e.g., ships, cars, trucks, etc..), 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.

The underline 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 CLS, whenever a *ship* approaches the harbor, it discovers the fragments provided by the *landing manager* and by the *gates*. These fragments model the harbor-specific procedures and regulations that the ship should execute for the landing. Different fragments may be provided by different gates to be used by certain types of ship. Similarly, the ship will publish its own fragments implementing the procedures to be followed for the unloading of cars.

Another important feature of the proposed framework is the possibility of leaving the handling of extraordinary/improbable situations to run time instead of analyzing all the extraordinary situations at design time and embedding the corresponding recovery activities in the business process. This kind of modeling extremely simplifies the specification of business processes that have to operate in dynamic environments, 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).

These dynamic features offered by the framework rely on a shared *context model*, describing 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 car, status of a car, availability of a storage area). 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., current location of car is storage area A), but also as a result of exogenous changes (e.g., car status is damaged, storage area unavailable). A *context configuration* is a snapshot of the context at a specific time, capturing the current status of all its context properties. For instance, the *CarLocation* diagram (depicted in Figure 2) captures how the car location can change over time. Initially, the car is on the ship. The car process aims to unload the car to the unpacking area and move it to the storage. The treatment location is where the car can be repaired. Similarly, the *CarStatus* diagram represents car operability status. An example of an exogenous change could be where the car status changes from *ok* to *nok* by the exogenous event *damaged*.

Business processes and fragments are modeled as *Adaptable Pervasive Flows* (APFs) [5], [13], 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), APFs 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 precondition *P1: CarStatus=ok and CarRegistration=no* in the *Registration Reply* activity of Figure 1B. The *Storage Manager*, provider of the fragment, specifies through this condition that the activity can be executed only if the car is not yet registered and is not damaged. The next activity of the same fragment, *Registration Reply*, is annotated with the effect *E1: CarRegistration.registered*, meaning that the expected impact of this activity is to make the system context evolve to a configuration where property *CarRegistration* is in state *yes*.

Figure 1: Adaptation mechanisms in the car delivery scenario

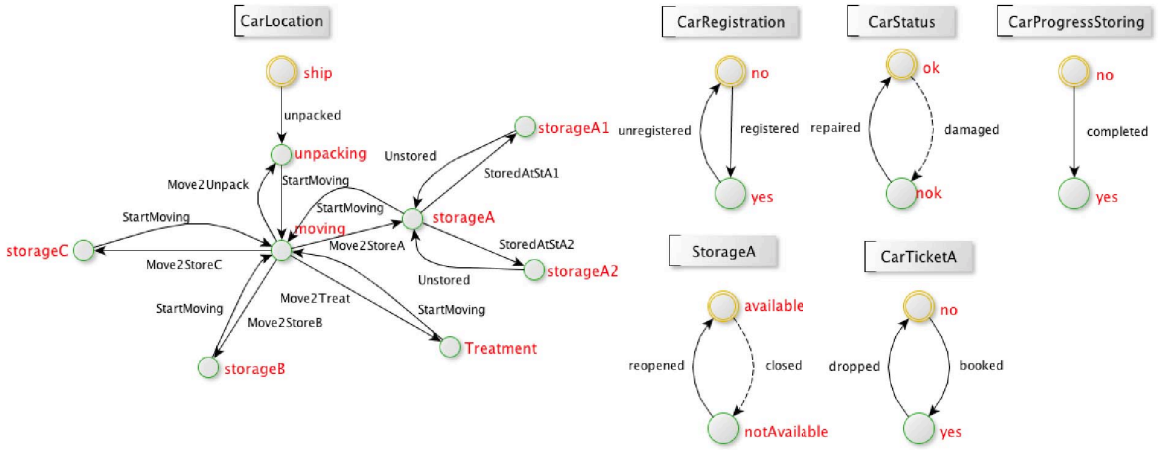

Figure 2: Examples of context properties for the CLS.

Finally, in order to have dynamic processes, we extended the APF language with constructs enabling the customization and adaptation of process fragments. In particular, we introduced the possibility of specifying *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 *Store* of the car process model in Figure 1A, aiming at storing the car in a storage area, is annotated with the goal *G3: CarProgressStoring=yes*. At run time, a specific fragment composition will be generated to achieve this goal taking into account the characteristics of the car, the status of the storage areas, and the available fragments for the car storing. 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. An example is the compensation goal *C1: CarRegistration=no* for the activity *Registration Reply* defined in the *Registration* fragment provided by the *Storage Manager* (Figure 1B), or *C2: CarTicketA=no* for the activity *BookA* defined in the fragment *StoreToA* provided by *Storage Area A* (Figure 1C).

B. Adaptation mechanisms

In this Section we present different adaptation mechanisms that can be used to handle the dynamicity of context-aware pervasive systems. Our framework can deal with two different adaptation needs: the need for refining an abstract activity within a process instance, and the violation of the context precondition of an activity that has to be executed. In the second case, the aim of adaptation is to solve the violation by bringing the system to a situation where the process execution can be resumed.

1) Refinement mechanism: 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 fragments provided by other entities in the system 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 truck arriving at the delivery area may provide its own loading fragment), and the composition strongly depends on the current execution context (e.g., a storage area may be full and thus its fragment not usable).

Consider, for instance, the abstract activity *Store* of the main car process in Figure 1A. During the execution the activity is automatically refined and composes five available fragments (i.e., Registration, StorageAssignment, StoreToA, StoreToB and StoreToC) provided by different entities (i.e, *Storage Manager*, *Storage Area A*, *Storage Area B*, and *Storage Area C*). The refinement obtained is injected in the car 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. The result of this incremental refinement is a multi-layer process execution model (see Figure 1), where the top layer is the initial process of the entity and intermediate layers correspond to incremental refinements.

2) Local Adaptation Mechanism: Local adaptation 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 1B. The car process is ready to execute the *Registration Reply* activity of the *Registration* fragment, however the car gets damaged and the precondition *P1* of the activity is not valid. The aim of local adaptation in this case is to repair the car (i.e., property *CarStatus=ok* must hold). This is achieved by composing two fragments that allow us to move the car to the treatment station (*MoveToTreatment*) and to repair it

(*Repairment*). After executing the local adaptation process, the car process instance can resume the execution of the original process.

3) Compensation Mechanism: 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.

Consider for instance the compensation of the *BookA* activity of the *StoreToA* fragment provided by *Storage Area A* in Figure 1C. The compensation goal *C2* associated to the activity requires that a context configuration where there are no places booked for the car in the storage area is reached. In our case the activity needs to be compensated after its completion and the generated compensation process requires that the ticket for the storage area is dropped.

C. Adaptation strategies

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 (e.g., there is no way of making the storage area A1 available for adaptation need *A3* of Figure 1E). Another example is the failure 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 mechanims. 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 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. All the patterns can be implemented through one-shot or incremental adaptation, and, if needed, other patterns can be easily added to the framework.

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 labelled 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 1 to solve the adaptation need *A3*, where the storage area A1 becomes unavailable. In this specific execution context, the re-refinement of the abstract activity *StoreAndDropA* requires to compensate the activities *BookTicketA1Reply* of the *BookStorageA1* fragment of entity *Storage Area A1* by dropping the ticket and then to re-compute a fragment composition that, taking into account the current storage availability, allows the care to be parked in Storage Area A2 (see Figure 1F).

The *backward adaptation* strategy 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 rollebacked (compensation mechanism), and for bringing the system to a configuration where the precondition of the activity to be executed is satisfied (local adaptation). This strategy is used in our scenario to deal with adaptation *A2* (see Figure 1D), where the storage area A is no longer available but a parking ticket has already been booked for the car. In this case, where neither local adaptation nor re-refinement would work, a successfull strategy could be to bring back the execution to an activity that can potentially make a different decision about the storage area assigned to the car (i.e., *AssignStorageRequest* in *StorageAssignment* fragment). To implement this strategy there is the need for compensating the *BookA* abstract activity by dropping the ticket (i.e., *DropTicketA* fragment) and of making the precondition *P2* valid (i.e., executing *Move2Unpacking* fragment, see Figure 1G).

Other strategies can be defined by composing the adaptation mechanisms in a different way by combining sub-strategies. To give an example, a possible strategy could be to search for a local adaptation, and, in case no solution is found, try backward adaptation within the same fragment composition, then apply the re-refinement mechanism, then recursevely apply backward and re-refinement mechanism, moving up in the execution layers, till the beginning of the process instance. A completely different strategy could be to search for alternative solutions in parallel and then choose the best solution according to a set of pre-defined metrics (e.g., number of activities to be performed, impact on the process structure, impact on the context configuration).

IV. Formal Framework

In this section we introduce formal definitions of the elements of our adaptation framework. Then, we show how the different adaptation mechanisms can be defined in terms of a *general adaptation problem*, and that any general adaptation problem formalized in our framework can be automatically resolved using planning techniques.

A. Definitions

1) Context Property: Every context property is modelled with a *context property diagram*, which is a state-transition system capturing all possible property values and value changes. Each transition is labeled with the corresponding *context event*.

Definition 1: (Context Property Diagram) Context property diagram is a tuple $c = \langle L, L_0, E, T \rangle$, where:

- L is a set of context states and $L_0 \subseteq L$ is a set of initial states;
- $E = E_{unc} \cup E_{cnt}$ is a set of context events, where E_{unc} is a set of uncontrollable and E_{cnt} is a set of controllable events, such that $E_{cnt} \cap E_{unc} = \emptyset$;
- $T \subseteq L \times E \times L$ is a transition relation.

We denote with $L(c)$, $E(c)$, etc. the corresponding elements of context property diagram c.

The overall context is usually quite complex and requires a set of context property diagrams C to be specified. In this case, a context state is a combination of states of its property diagrams. Formally a space of *context configurations* is defined as $L = \prod_{c \in C} L(c)$. We also define $E_{cnt} = \bigcup_{c \in C} E_{cnt}(c)$.

2) Process Fragments: 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.

As described in Section III-A, activities can be annotated with preconditions, effects, and compensations, while abstract activities are annotated with goals. In the definition of process fragments, this is captured through dedicated labeling functions.

Definition 2: (Process Fragment) Process Fragment defined over context property diagrams C 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 activitiess, 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;
- $Ann = \langle Pre, Eff, Goal, Comp \rangle$ is a process annotation, where $Pre : A_{in} \cup A_{out} \cup A_{con} \rightarrow L^*$ is the precondition labeling function, $Eff : A_{in} \cup A_{out} \cup A_{con} \rightarrow E^*_{cnt}$ is the effect labeling function, $Goal : A_{abs} \rightarrow L^*$ is the goal labeling function, and $Comp : A \rightarrow L^*$ is the compensation labeling function.

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

3) Process and System Executions: The following definition captures the current status of the execution of a given process. As illustrated in Figure 1, the process is a hierarchical structure, obtained through the refinement of abstract activities into fragments. In the following definition, a process configuration is hence 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 3: (Process Configuration) We define a process configuration as a non-empty list of triples $E_p = (p_1, a_1, h_1), (p_2, a_2, h_2) \ldots (p_n, a_n, h_n)$, 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 4: (System Configuration) Given a set C of context property diagrams, we define a system configuration for C as a tuple $\mathcal{S} = \langle \mathcal{I}, \mathcal{E}, \mathcal{F} \rangle$, where:

- $\mathcal{I} \in L(c_1) \times \ldots \times L(c_n), c_i \in C$ 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.

B. Adaptation

1) General Adaptation Problem: In the following we define the concept of general adaptation problem, which will be used to formalize the different adaptation mechanisms presented in Section III-B.

A general adaptation problem contains complete information about the system configuration and an adaptation goal, which captures the adaptation objectives to be fulfilled.

Definition 5: (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 C.
We denote with $\mathcal{S}(\xi)$ and $\mathcal{G}(\xi)$ the corresponding elements of an adaptation problem ξ.

For expressing the adaptation goals, we exploit EAGLE [9], an expressive language introduced to define complex goals in the scope of automated planning. EAGLE is particularly suited to be used to model adaptation problems, and in particular adaptation strategies, since it allows the definition of: (i) goals as sets of context configurations, $\mathcal{G} \subseteq L$, and (ii) sequential composition of goals $\mathcal{G} = \mathcal{G}_1 \cdot \mathcal{G}_2$, defining a temporal order in which the sub-goals need to be satisfied by the adaptation.

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) From Adaptation Strategies to 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 $E_p = (p_1, a_1, h_1), (p_2, a_2, h_2), \dots (p_n, a_n, h_n)$, 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 $E'_p = (p'_1, a_0, h_1), (p_2, a_2, h_2), \dots (p_n, a_n, h_n)$ 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 $h_1 = h_0, \tilde{a}_1, \dots, \tilde{a}_m$, and that we want to rollback activities $\tilde{a}_1, \dots, \tilde{a}_m$. 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 rollbacked. $\mathcal{S}'$ is obtained from $\mathcal{S}$ by updating E_p into $E'_p = (p'_1, a_0, h_0), (p_2, a_2, h_2), \dots (p_n, a_n, h_n)$, where p'_1 is obtained by merging M_{adapt} into p_1 before activity $\tilde{a}_1$.

Adaptation Strategies. As described in Section III-C, 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 $\xi_1 \cdot \xi_2 = \langle S, \mathcal{G}_1 \cdot \mathcal{G}_2 \rangle$. We remark that, in the *one shot* case, the definition of $\mathcal{S}'$ depends on the specific strategy: for instance, in the case of a backward adaptation, if the system configuration has to be brought back to activity a_k of fragment p_k, with $k > 1$, then all elements $(p_1, a_1, h_1), \dots (p_{k-1}, a_{k-1}, h_{k-1})$ are removed from the system configuration, and element (p_k, a_k, h_k) is suitably modified to include the adaptation.

3) Adaptation as AI Planning Problem: In order to automatically solve general adaptation problems, we adopt and adjust the service composition approach presented in [2][1]. 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.

A planning domain is derived from adaptive problem ξ. In particular, a set of n fragments $(p_1, \dots, p_n \in \mathcal{F}(\mathcal{S}))$, m context property diagrams $(c_1, \dots, c_m \in C)$ are transformed into state transition systems (STS) using transformation rules similar to those presented in [2]. 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 $\Sigma_{p_1} \dots \Sigma_{p_n}$ and $\Sigma_{c_1} \dots \Sigma_{c_m}$, 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.

$$\Sigma = \Sigma_{p_1} \| \dots \| \Sigma_{p_n} \parallel \Sigma_{c_1} \| \dots \| \Sigma_{c_m}$$

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 [17] 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.

Correctness of the approach. The proof of the correctness of the approach consists in showing that, under the aforementioned assumptions, all the executions of the adaptation process M_{adapt} (translation of controller Σ_c) implement the adaptation strategy. Here we outline the key points of the proof. It is easy to see that each execution θ of the adaptation process is also a run of the domain, i. e., if $\theta \in \Pi(\Sigma_c)$ then $\theta \in \Pi(\Sigma)$. Under the planning requirement that all the executions of the domain terminate in goal states, we get that all executions of the domain implement the adaptation strategy. As a consequence, the following theorem holds.

Theorem 1: (Correctness of the approach) Let:

- $\Sigma_{f_1}, \dots, \Sigma_{f_n}$ be the STS encoding of fragments $f_1, \dots f_n$ and
- $\Sigma_{c_1}, \dots, \Sigma_{c_m}$ be the STS encoding of context property diagrams $c_1, \dots c_m$.

Let Σ_c be the controller for a particular composition problem

$$\Sigma = \Sigma_{f_1} \| \dots \| \Sigma_{f_n} \parallel \Sigma_{c_1} \| \dots \| \Sigma_{c_m}$$

$$I(\Sigma) = r, \rho = G^{\Sigma}$$

[1]This approach to service composition deals with a broader set of features than the ones discussed in this paper. Prominently, it is able to manage data and data requirements in an efficient way [16]. For simplicity and lack of space, we did not consider data explicitly in the description of process fragments adopted in this paper. The selected composition approach, however, ensures that the adaptation approach presented in this paper also works with descriptions of process fragments that are enriched with data.

i.e., $\Sigma_c \triangleright \Sigma \models \rho$. Then execution $\Pi(\Sigma_c)$ implements the 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. Also, for the same adaptation problem, more than one solution may exist, especially if several fragments have similar or identical functionality. In this case, the planner returns the plan that is shortest with respect to the number of execution steps [18].

V. Evaluation

The proposed adaptation mechanisms and strategies have been implemented in the ASTRO-CAptEvo Framework [20], developed on top of the ASTRO project (www.astroproject.org). The framework exploits the modeling approach described in Section III-A where adaptability and context-awareness are key embedded characteristics of the application. We have created a visualization environment enabling interaction between the framework and the user and simulating execution and adaptation of business processes in our case study. In particular, it can (i) run the CLS and simulate the execution and adaptation of business process attached to entities and (ii) visualize in real time the execution and adaptation of business processes. The specification of the CLS we used to evaluate our approach contains 29 entity types, 69 process fragment models and 40 types of context properties. We evaluated our techniques using a dual-core CPU running at 2.8GHz, with 8Gb memory. We carried out an experiment in which we kept the demonstrator running for one hour, simulating the operation of the Bremen port with new ships loaded with cars constantly arriving. In parallel, we collected the execution and adaptation statistics such as the number of adaptation cases, the average number of fragments used to solve an adaptation problem, the average time to generate an adaptation solution and the number of total entities managed.

Within one hour, more than 1000 adaptation cases were generated. On average, an adaptation problem contained 7 process fragments (the fragments that could potentially be used in the final solution, and, as such, were presented in the planning domain), and took less than 4 seconds to be resolved. During the experiment, the framework was able to manage more than 350 entities (i.e., ships, cars, trucks, etc..).

These results demonstrated the usability and scalability of our approach when applied to a real-world complex scenario. To see the framework in action, when applied to the aforementioned CLS scenario, one can download the complete demo or the video of a live demonstration at: www.astroproject.org/captevo.

VI. Related Work and Conclusion

In this paper we have proposed an approach to 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 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.

Designing the adaptation of systems on the architecture level offers several potential benefits, such as the appropriate abstraction level to describe dynamic changes in a system, the potential for scalability to large-scale complex applications, and the generality that allows one to design solutions for a wide range of application domains [14].

Garlan et al. [7] introduce an SA-based self-adaptation framework, called Rainbow, which uses external mechanisms and an SA model to monitor a managed system, detect problems, determine a course of action, and carry out the adaptation actions. Rainbow, by making use of architectural styles, provides general and reusable infrastructures with explicit customization points. The definition of these customization points limits the dynamicity of the approach; in particular, the context is not considered as a part of the system model that can evolve during the system life-cycle. In our approach we do not rely on pre-defined customization points to manage the adaptation. Contrariwise, we monitor properties of the context to understand where and how adapt the system.

As mentioned in [12], in order to support services whose adaptive behavior can evolve over time, there is a need for higher level adaptation policies that are architecture independent. In this respect, Kramer and Magee propose in [14] a three-layer reference architecture for dealing with self-managed or self-* systems. The uppermost layer is the goal management that produces from a high level system goal, a plan to achieve the goal. This plan is the input of the change management layer that can introduce new components, recreate failed components, change component interconnections and so on. The bottom layer is the component control which ensures to preserve safe application operation during adaptation. In this approach, adaptations can only be triggered for the satisfaction of a declared goal. Our approach is more general, it is able to adapt the system not only when new requirements are defined (i.e., new goals), but also when the context model evolves and reaches a state that triggers the adaptation need.

In the community of Service Oriented Computing (SOC), various approaches supporting adaptation have been defined, e.g., triggering repairing strategies as a consequence of a requirement violation [22], and optimizing QoS of service-based applications [6], [25], [26], or for satisfying some application constraints [23]. Repairing strategies could be specified by means of policies to manage the dynamism of the execution environment [1], [8] or of the context of mobile service-based applications [21]. The aim of the strategies proposed by the aforementioned approaches range from service selection to rebinding and application reconfiguration [19], [24]. These are interesting features, but cannot deal with comples process restructuring and lack a design approach to support designers in developing applications where context-awareness and adaptivity are key characteristics. An adaptation approach similar to ours is presented in [10]. SmartPM is a formal framework based on situation calculus and IndiGolog language that provides run-time adaptation of a process to deal with unplanned exceptions. The problem of adaptation is eventually reduced to a classical planning prob-

lem. However, SmartPM is currently a theoretical framework and the efficiency and overhead of adaptation modelling 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 APF to state transition systems, provides efficient development tools that minimize the adaptation modelling effort.

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*). Another interesting research challenge we want to tackle is to learn from past adaptations what are the best strategies to be used to solve specific adaptation problems and use the result of this analysis to improve the strategies. We are also investigating the possiblity 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] 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.

[3] F. Böse and J. Piotrowski. Autonomously controlled storage management in vehicle logistics applications of RFID and mobile computing systems. *International Journal of RT Technologies: Research an Application*, 1(1):57–76, 2009.

[4] A. Bucchiarone, R. Kazhamiakin, M. Pistore, and H. Raik. Adaptation of service-based business processes by context-aware replanning. In *SOCA 2011*, 2011.

[5] A. Bucchiarone, A. Lluch Lafuente, A. Marconi, and M. Pistore. A formalisation of Adaptable Pervasive Flows. In *WS-FM'09*, Bologna, Italy, 4 September 2009.

[6] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of GECCO '05*, pages 1069–1075, 2005.

[7] S. Cheng, A. Huang, D. Garlan, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In *ICAC*, pages 276–277, 2004.

[8] 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.

[9] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a Language for Extended Goals. In *Proc. AAAI'02*, 2002.

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

[11] H. Eberle, T. Unger, and F. Leymann. Process fragments. In *On the Move to Meaningful Internet Systems: OTM 2009*, volume 5870 of *Lecture Notes in Computer Science*, pages 398–405, 2009.

[12] E. Gjørven, F. Eliassen, K. Lund, V. S. Wold Eide, and R. Staehli. Self-adaptive systems: A middleware managed approach. In *2nd IEEE International Workshop on Self-Managed Networks, Systems & Services (SelfMan 2006)*, 2006.

[13] K. Herrmann, K. Rothermel, G. Kortuem, and N. Dulay. Adaptable Pervasive Flows - An Emerging Technology for Pervasive Adaptation. In *Workshop on Pervasive Adaptation (PerAda)*. IEEE Computer Society, September 2008.

[14] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*, FOSE '07, pages 259–268, 2007.

[15] A. Marconi, M. Pistore, A. Sirbu, H. Eberle, F. Leymann, and T. Unger. Enabling adaptation of pervasive flows: Built-in contextual adaptation. In *ICSOC/ServiceWave*, pages 445–454, 2009.

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

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

[18] D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

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

[20] H. Raik, A. Bucchiarone, N. Khurshid, A. Marconi, and M. Pistore. Astro-captevo: Dynamic context-aware adaptation for service-based systems. In *SERVICES*, 2012.

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

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

[23] 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.

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

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

[26] 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.