# On-the-Fly Adaptation of Dynamic Service-Based Systems: Incrementality, Reduction and Reuse

Antonio Bucchiarone, Annapaola Marconi,
Claudio Antares Mezzina, Marco Pistore, and Heorhi Raik

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

**Abstract.** On-the-fly adaptation is where adaptation activities are not explicitly represented at design time but are discovered and managed at run time considering all aspect of the execution environments. In this paper we present a comprehensive framework for the on-the-fly adaptation of highly dynamic service-based systems. The framework relies on advanced context-aware adaptation techniques that allow for i) incremental handling of complex adaptation problems by interleaving problem solving and solution execution, ii) reduction in the complexity of each adaptation problem by minimizing the search space according to the specific execution context, and iii) reuse of adaptation solutions by learning from past executions. We evaluate the applicability of the proposed approach on a real world scenario based on the operation of the Bremen sea port.

## 1 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 [13], 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.

Despite the considerable effort dedicated in recent years to investigate approaches for the adaptation of service-based systems, we are still far from effective solutions. As we will discuss in depth in the related work, 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, and can hardly be used in dynamic settings; or only deal with very limited forms of "local" dynamic adaptation, e.g., service replacement.

In recent work [4], we have proposed a comprehensive framework for the on-the-fly adaptation of service-based application. This approach exploits the concept of process fragments [9] as a way to model reusable process knowledge and to allow for an *incremental* and context-aware composition of such fragments into adaptable service-based

S. Basu et al. (Eds.): ICSOC 2013, LNCS 8274, pp. 146–161, 2013.
© Springer-Verlag Berlin Heidelberg 2013

applications. The framework allows for 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 supports on-the-fly adaptation to unexpected or improbable context changes that may affect the execution of the application. Also in this case, available fragments and current context are exploited to solve the problem.

Since highly dynamic systems generates a large number of adaptation problems (both in terms of process refinements and of other forms of adaptation), this paper investigates a potential problem of on-the-fly adaptation for such system: is it feasible to solve large number of adaptation problems at run time, each involving a potentially very large set of other actors and available fragments? We answer to this question by extending the framework of [4] in two directions: first, we show that it is possible to *reduce* the complexity of each adaptation problem by minimizing the search space so that it only includes fragments and properties that are relevant for the problem; second, we show that it is possible to store and *reuse* previously discovered adaptation solutions, thus learning from past executions and reducing the number of adaptations to be effectively computed at run time. These two extensions are complementary and integrated: by reducing the adaptation problems to their minimal versions, we increase the number of adaptation problems that turn out to be equal, and we hence increase the possibility of reusing previous solutions.

We evaluate the proposed approach on a real world scenario based on the operation of car logistics in the Bremen sea port [3]. We show that it is effective in reducing the number of requested adaptations: the experiments show that, while the situation where new adaptation are not needed is never reached (thus witnessing the need of dynamic adaptation), the number of such new adaptations decreases over time (thus making reuse more and more efficient). We also show that the approach seamlessly accommodates situations where previous adaptations are not valid anymore, e.g., due to changes in the requirements or in the available fragments: these changes are reflected in changes in the context that make the past solutions not reusable; the approach computes new solutions suited for the new requirements and fragments; and the overhead in terms of performance of this computation of new plans is very limited.

The rest of the paper is structured as follows. Following, we shortly introduce the Bremen harbor car logistic scenario that is used as a reference throughout the paper. Section 2 presents the proposed framework for on-the-fly adaptation; Section 3 gives a formal specification of the framework, while Section 4 shows the definition and implementation of adaptation reduction and reuse within the framework. Finally, Section 5 describes how we have evaluated our solution using the car logistic demonstrator while Section 6 presents some related works, and conclusions.

### 1.1 Motivating Scenario: Car Logistics

The reference scenario 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*.

Our goal is to develop a system (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, both in terms of the variability of the actors' involved and of their procedures (customizable processes), and of the exogenous context changes affecting its operation.

Considering for instance the delivery process of each car, customization means that the delivery procedure of each car needs to be customized according to the car brand, model, retailer-specific requirements, etc. Moreover new car models, having specific requirements and procedures, have to be able to be easily integrated in the system. Similarly, the system needs to flexibly deal with changes in the procedures of external actors such as ships and trucks. Concerning *context dynamicity*, examples of environment conditions to be taken into account are the unavailability or malfunctioning of the different port facilities, accidental damages of cars and trucks, human errors (e.g., a car is parked in the wrong parking lot).

## 2 General Framework and Approach

In this section we present our framework for creating and running adaptive context-aware service-based systems like the CLS described above.

### 2.1 Modeling Artifacts

In our framework, we model the real-world system under consideration as a set of *entities* that can collaborate with each other in order to accomplish their business goals (e.g., in the CLS scenario such entities might be cars, ships and other port facilities). In turn, the entity model includes 1) *entity context* capturing key characteristics of the entity, 2) business *process* determining entity behaviour and 3) a set of *process fragments* (from now on, simply *fragments*) that can be exploited by external partners (i.e., other entities) in order to collaborate with the fragments owner.

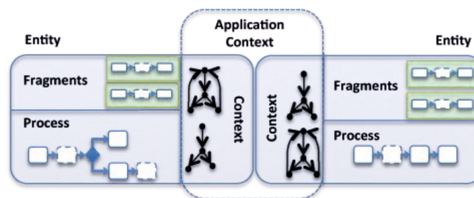


**Fig. 1.** Modeling Artifacts

Entity context is modeled as a set of *context properties*, each capturing some relevant characteristic of an entity (e. g., car location, car status, etc.). A context property is represented by a context property diagram, a state transition system containing all possible property values (states) and value changes (transitions labeled with events).

For instance, car location may be changed from *storage* to *mechanical station* when the car moves around the harbour area. The overall application context is composed of context properties of all constituent entities (see Figure 1).

Entity process and fragments are modeled as Adaptable Pervasive Flows (APFs) [5], that is an extension of classical workflow languages that 1) adds the possibility to model special types of process activities (most interestingly, *abstract activities*), and 2) introduce contextual annotations that connect processes to their operational context. Abstract activities let us include in a process some task whose actual implementation cannot be efficiently provided at design time and needs to be dynamically determined (or generated) at run time. Contextual annotations include preconditions, effects, goals and compensations. Activity *precondition* shows in which contextual situations (states) the activity execution is allowed. Activity *effect* indicates which contextual events are triggered when the activity is executed. Abstract activity *goal* expresses an abstract task associated with the abstract activity in terms of goal contextual situations. Finally, activity *compensation* specifies how the activity can be compensated after execution (similarly to goals, it is expressed as a set of context situations in which activity effects are considered to be compensated).

The proposed set of modeling artifacts is able to capture the key characteristics of dynamic context-aware systems since i) abstract activities expressed in terms of goals allow for run-time selection of fragments according to their availability and further fragment composition in compliance with the execution context, ii) the context-awareness of processes allows us to detect execution problems at run time (e. g., by detecting precondition violations) and produce solutions to them dynamically (using fragment composition tools), iii) entities can join/leave the system at run time without interrupting its operation.

### 2.2 On-the-Fly Adaptation Approach

Our approach enables on-the-fly adaptation of context-aware systems by combining the four key features: i) incremental resolution of complex adaptation problems by interleaving problem solving and solution execution, ii) reduction in the complexity of each adaptation problem (by using the search space that contains only information that is relevant for a given problem and context), iii) reuse of adaptation solutions by learning from past executions, and iv) exploitation of advanced AI planning techniques [2] to solve adaptation problems by appropriately composing available fragments. The general idea of the approach consists in constantly monitoring process execution, detecting various forms of inconsistencies (for now, these includes unrefined abstract activities and precondition violation) and resolving them through composition of available fragments. In Figure 2 we show the approach life-cycle and in the rest of this section we explain it in detail.

**Incremental Adaptation.** As described in Section 2.1, a key feature of the framework is the possibility of partially specifying the process logic at design-time, leaving the refinement of abstract activities and resolution of most problems to run time. The advantage of performing process adaptation at run time is twofold. First, efficient adaptation heavily depends on run-time status of the execution environment (e.g., on the

set of available fragments and actual context), which often is unknown at design time. Second, availability of automated adaptation tools significantly simplifies the work of process designer, who now does not need to consider all special cases and to implement all tasks at design time.

Abstract activity refinement consists in producing fragment composition that satisfies abstract activity goal and thus can be used as an activity implementation. Since fragments used in refinement may also contain abstract activities, the resulting process instance has a multi-layer structure, where the top layer is the initial process and intermediate layers correspond to incremental refinements. Consider, for instance, the abstract activity *Store* of the main car process in Figure 3. During the execution, the activity is automatically refined and composes four available fragments (i.e., Registration, StorageAssignment, StoreToA, StoreToB ) provided by different entities (i.e, *Storage Manager*, *Storage Area A*, *Storage Area B*). The abstract activity *BookA* within this refinement is further refined with fragments provided by *StorageAreaA* entity.

Considering another form of on-the-fly adaptation (i. e., reaction to precondition violation), the aim of the composition produced is to bring the system to a context where the process execution can be resumed (i.e., precondition is not violated anymore). This is the case of adaptation *A1* in Figure 3, where the car has been damaged and violates the precondition of the *Registration* fragment. The framework supports different adaptation mechanisms to tackle this problem, among which: *local adaptation*, where the aim is to bring the system to a context configuration satisfying the violated precondition; *on-the-fly compensation*, that can be used to dynamically compute a compensation process for an activity or a set of already executed activities; *re-refinement*, combining

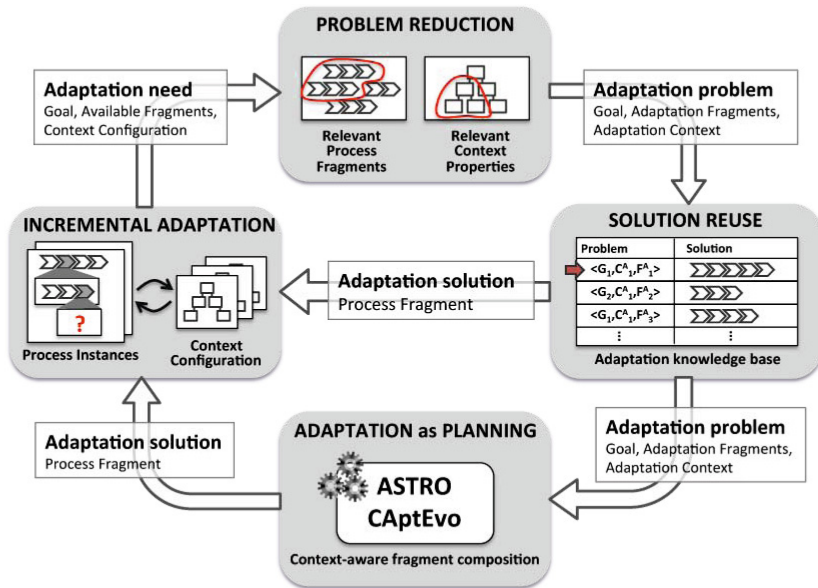


**Fig. 2.** Overview of the On-the-Fly Adaptation Approach

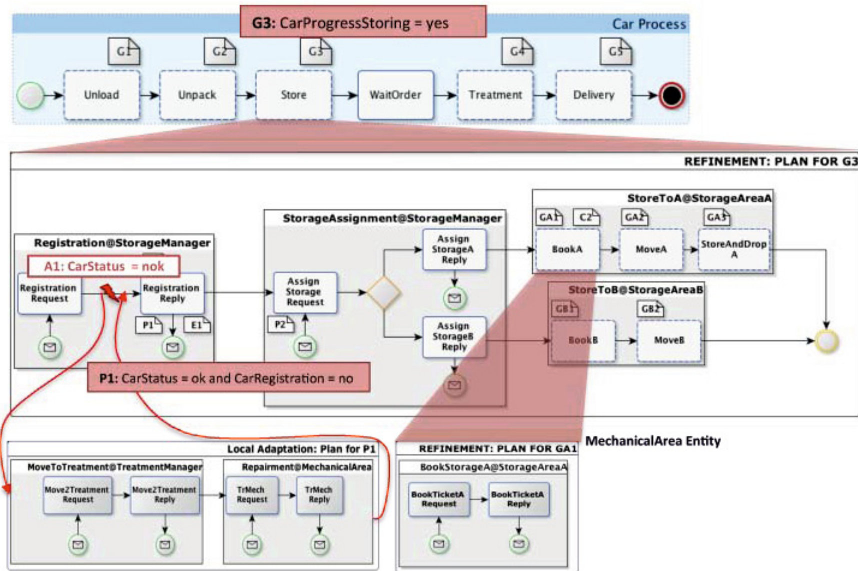


**Fig. 3.** Process Adaptations in action

compensation and refinement mechanisms to re-compute the refinement of an abstract activity considering the new execution context. A complete description and definition of all the adaptation mechanisms and strategies provided by the ASTRO-CAptEvo Framework is presented in [4].

To summarize, this phase can deal with two different *adaptation needs*: the need for refining an abstract activity and the need to resolve precondition violation. In both cases, an adaptation problem is formally specified, in terms of the goal to be reached, the available fragments, and the current context configuration. Then, it is passed as input to the problem reduction phase.

**Problem Reduction.** The aim of this phase is to optimize an adaptation problem in order to reduce search space for further planning. The point is that while the whole system can be rather complex (including dozens of facilities and thousands of cars), only its small portion is relevant for a particular inconsistency resolution (e. g., if we need to plan car unloading, we need to consider only this particular car and only the actors participating in unloading). This is done in two steps. First, we identify the range of entities (participants) that are relevant for the adaptation problem in hands. Second, we reduce preselected fragments and context properties taking into account the current context state of the system (e.g., remove all transitions and state that can never be reached from the current state). These two steps are further explained in Section 4.1.

It is worth to mention that the reduction phase is key not only for the *Adaptation as Planning* phase, but also for the *Solution Reuse* phase, since it allows for better characterization of the adaptation problem.

**Solution Reuse.** Given an adaptation problem, this phase checks whether absolutely the same problem has already been solved in the past. Though simple from a conceptual

point of view, this step requires generalization of a specific adaptation problem so that it is abstracted away from specific instances and can be conceptually compared to similar problems previously resolved (see Section 4.2 for the details). If a solution exists, it is properly grounded to the instance-level adaptation problem and passed to the *Incremental Execution* phase to be executed. Otherwise, the adaptation problem is passed to the *Adaptation as AI Planning* phase.

**Adaptation as AI Planning.** This phase is responsible for finding a solution to the adaptation problem (i.e., a new fragment), by automatically composing the set of available fragments, according to the current context configuration and to the goal to be achieved. This phase exploits the ASTRO-CAptEvo adaptation engine [14] that transforms an adaptation problem into a planning problem and applies to it advanced planning techniques capable of dealing with asynchronous nondeterministic domains and complex goals ([2,10]).

## 3 Formal Framework: Background

In this section we introduce formal definitions of the core elements of our adaptation framework. They will be used in Section 4 to present our solution.

### 3.1 Elements

**Definition 1 (Context Property Type).** *A context property type is a state transition system* $c = \langle L, l^0, E, T\rangle$*, where:*

- $L$ *is a set of context states and* $l^0 \in L$ *is the initial state;*
- $E = E_{unc} \cup E_{cnt}$ *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.*

The *context model* of a system is composed by a set of context property types $C_M = \{c_1, \ldots, c_n\}$ such that $c_i = \langle L_i, l_i^0, E_i, T_i\rangle$ and $L_i \cap L_j = \emptyset$ and $E_i \cap E_j = \emptyset$ if $i \neq j$. For each context property type $c_i$ there may exist zero or more instances at run time, hence we define the *runtime context state* as a set of the states of the all the instances.

**Definition 2 (Runtime Context State).** *A runtime context state is a set* $C = \{(l_{i,j})\}$ *such that* $l_{i,j} \in L_i$ *for some* $c_i = \langle L_i, l_i^0, E_i, T_i\rangle \in C_M$.

Since we want to relate multiple states of the same context property $c_i$ with different instances, we define the set of all possible states of an instance $j$ of type $i$ as the set $L_j^i = \{(j, l) \mid l \in L_i\}$.

We denote with $\mathbb{L} = (\prod_{\forall i | c_i \in C_M} L_i)$ and $\mathbb{L}_C = (\prod_{\forall i \forall j | l_{i,j} \in C} L_i^j)$.

Set $\mathbb{L}$ represents the set of all the possible configurations (in terms of states) in which the context model $C_M$ can be, while $\mathbb{L}_C$ represents the set of all the possible configurations in which the runtime context $C$ can be. In the same way sets $\mathbb{E}$ and $\mathbb{E}_C$ are

defined, representing respectively all the possible combinations of events of the model and of the runtime context.

Processes (and fragments) are modeled as state transition systems, where each transition corresponds to a particular process 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. In the following we will indicate with $A^\star$ either the set $A$ or $\emptyset$. Abstract activities can be annotated with goals, while input, output and concrete activities can be annotated with preconditions, effects, and compensations. We define a *process instance* as follows:

**Definition 3 (Process Instance).** *A process instance defined over the runtime context state $C$ 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 activities, $A_{con}$ is a set of concrete activities, and $A_{abs}$ is a set of abstract activities. $A_{in}$, $A_{out}$, $A_{con}$, and $A_{abs}$ are disjoint sets;*
- *$T \subseteq S \times A \times S$ is a transition relation;*
- *$Ann = \langle Pre, Eff, Goal, Comp\rangle$ is a process annotation, where $Pre : A_{in} \cup A_{out} \cup A_{con} \rightarrow \mathbb{L}_C^\star$ is the precondition labeling function, $Eff : A_{in} \cup A_{out} \cup A_{con} \rightarrow \mathbb{E}_C^\star$ is the effect labeling function, $Goal : A_{abs} \rightarrow \mathbb{L}_C$ is the goal labeling function, and $Comp : A \rightarrow \mathbb{L}_C^\star$ is the compensation labeling function;*

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

**Definition 4 (Process Fragment).** *A process fragment defined over the context model $C_M$ is a tuple $f = \langle S, s^0, A, T, Ann, En\rangle$, where:*

- *$S, s^0, A, T$ are as Definition 3;*
- *Ann is as Definition 3 but on set $\mathbb{L}^\star$ and $\mathbb{E}^\star$*
- *$E_n$ is a set of entity types that can use the process fragment $f$.*

Let $a \in A(f)$ an activity of a process fragment $f$ and $l = Ann(a) \subset \mathbb{L}^\star$ its annotation. To understand if $l$ contains states of a certain context property type $c_i \in C_M$ we define the projection of the annotation $l$ onto context property type $c_i$ as $l \downarrow_{c_i} = l_i$. In the same way we define an instance based projection on $\mathbb{L}_C$, written $\mathbb{L}_C \downarrow_{i,j}$ to capture all the possible states of an instance $j$ of type $c_i$.

As described in Section 2, the system operation is modeled through a set of entities (e.g., ships, cars, trucks, etc..), each specifying its behavior through a process and offering their services through a set of process fragments. Formally an entity type is defined as:

**Definition 5 (Entity Type).** *An entity type $\mathcal{E}$ is a tuple $\mathcal{E} = \langle p, \mathcal{F}, C_\mathcal{E}\rangle$ where $p$ is the entity behaviour (i.e., process), $\mathcal{F}$ is a set of process fragments provided by the entity and $C_\mathcal{E} \subseteq C_M$ is a set of context property types that characterize the entity itself (i.e., CarLocation, etc..). We denote with $p(\mathcal{E})$, $\mathcal{F}(\mathcal{E})$ the corresponding elements of an entity $\mathcal{E}$.*

### 3.2 Execution Model

As illustrated in Section 2, an adaptable process is a multi-layer process, where the top layer is the initial process and the intermediate layers correspond to the adaptations (i.e, incremental refinements, local adaptations and compensations). For this reason we model the *process execution* of an adaptable process as a stack of pairs process-state. In the pair, *process* is a fragment as defined in Definition 4, while *state* is the current state of the process instance. The bottom (first) pair refers to the core process and all the others refers to adaptation processes. The top (last) pair in the stack is the one that is currently under execution. Pairs can be pushed to the stack when process adaptation is performed and can be popped from the stack when, e.g., the process instance of the top pair terminates.

**Definition 6.** *(Process Execution) A process execution is a non-empty stack of pairs* $\phi = (p_1, s_1), (p_2, s_2), \ldots, (p_n, s_n)$*, where:* $p_i = \langle S_i, s_i^0, A_i, T_i, Ann_i, E_{n_i} \rangle \in \phi$ *is a process fragment, while* $s_i \in S_i$ *is the current state in the corresponding process fragments.*

Following Definition 5, we define *entity instance* as follows:

**Definition 7 (Entity Instance).** *An entity instance e of type* $\mathcal{E}$ *is a tuple* $e = \langle p_e, \mathcal{F}, C_e \rangle$ *where* $p_e$ *is an instance of the process* $p(\mathcal{E})$*,* $\mathcal{F} = \mathcal{F}(\mathcal{E})$ *is a set of process fragments provided by the entity and* $C_e = \{(i, j)\}$ *s.t.* $c_i \in C_{\mathcal{E}}$ *and* $l_{i,j} \in C$ *is a set of pairs indicating the context property instances that characterize the entity instance.*

The running configuration of the whole system is defined by the runtime context state, by the process instances in the system, and by the set of available fragments.

**Definition 8.** *(Running System Configuration) A running system configuration is a tuple* $\mathcal{S} = \langle C_M, \mathcal{F}, C, \Omega \rangle$*, where:* $C_M$ *is the context model,* $\mathcal{F}$ *is the set of fragments available in the system,* $C$ *is the runtime context state, and* $\Omega$ *is a set of pairs* $(p_i, \mathcal{E}_j)$ *where* $p_i$ *is a process instance of* $p(\mathcal{E}_j)$*.*

### 3.3 Adaptation Need and Solution

Our framework can deal with two different adaptation needs [4,14]: 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. The *refinement adaptation* 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

An adaptation need captures all the runtime system information at the time of the violations. We formalize it as follows:

**Definition 9 (Adaptation Need).** *An adaptation need is a tuple* $\xi = \langle C_M, \mathcal{F}, C, \mathcal{G}, (p_i, \mathcal{E}_j) \rangle$*, where:* $C_M$ *is the context model,* $\mathcal{F}$ *is a set of process fragments available in the system annotated over context model* $C_M$*,* $C$ *is the runtime context state,* $\mathcal{G}$ *is an adaptation goal over* $C$*, and* $(p_i, \mathcal{E}_j)$ *is the process instance* $p_i$ *that needs to be adapted and* $\mathcal{E}_j$ *is the type of the entity to which the process instance belongs.*

We denote with $C(\xi)$ and $\mathcal{F}(\xi)$ the corresponding elements of an adaptation need $\xi$.

For expressing the adaptation goals, we exploit EAGLE [7] that allows the definition of goals as sets of context configurations, $\mathcal{G} \subseteq C$.

An *adaptation solution* is a process fragment $f_{adapt}$ that is obtained as the composition of a set of fragments in $\mathcal{F}$. When executed from the current system configuration $\mathcal{S}$, and in absence of exogenous events corresponding to unpredicted situations, $f_{adapt}$ ensures that the resulting runtime context state $C$ satisfies the goal $\mathcal{G}(\xi)$ of the adaptation need $\xi$.

# 4 Formal Framework: Solution

The aim of this section is to present how the on-the-fly adaptation framework works. Starting from the elements introduced in the previous section we present: (i) how an adaptation problem is generated from an adaption need, (ii) how an adaptation problem is optimized, and (iii) how we can reuse or find a solution for that problem.

## 4.1 Adaptation Problem

With respect to an adaptation need, an *adaptation problem* captures all the relevant system information needed to resolve it. In our approach it is generated by calling first the function *reduce* of Figure 4 and then function *optimize* of Figure 5.

The *reduce* function takes as input an adaptation need $\xi$ and returns a first version of an adaptation problem with the set of relevant fragments $\mathcal{F}_{rd}$ and context properties $C_{rd}$ that can be used to satisfy the need. It is computed in two steps: **Step 1** (lines 3-5) analyzes each fragment $f \in \mathcal{F}$ and selects only those that can be used by the entity $e_i$ ($e_i \in E_n(f)$); **Step 2** (lines 6-15) analyzes each fragment $f \in \mathcal{F}_{rd}$ and for each activity $a$ that belongs to $f$ it retrieves its annotations $l$ (line 8). From all the context properties instances $l_{i,j} \in C$ the algorithm first selects only those whose type is part of the annotation $l$ and that are defined by the entity $e_i$ (lines 10-11). Afterwards, it selects all the context property instances that are used by the fragments provided by the entities that eventually can collaborate with $e_i$ (lines 13-15).

Once a reduced version of the adaptation problem is obtained, we further optimize it by eliminating all states (transitions) in the context properties and process fragments that a priori will never be reached (triggered). The optimization algorithm is shown in Figure 5. The main function *optimize* (lines 27-40) takes as input an adaptation problem and returns its optimized version. The function repeats two optimization steps (lines 32-39) until the fixed point is reached.

```
1  function reduce(⟨C_M, 𝓕, C, 𝒢, (p_i, e_i)⟩)
2    𝓕_rd = ∅; C_rd = ∅; C_{M_rd} = C_M;
3    foreach(f∈ 𝓕)
4      if(e_i ∈ E_n(f))
5        𝓕_rd = 𝓕_rd ∪ {f};
6    foreach(f∈ 𝓕_rd)
7      foreach(a∈ A(f))
8        l = Ann(a);
9        foreach(l_{i,j} ∈ C)
10         if (l ↓_{c_i} ∈ l ∧ (i, j) ∈ C_e(e_i))
11           C_rd = C_rd ∪ {(l_{i,j})};
12         else C_{M_rd} = C_{M_rd} \ {c_i}
13         foreach (ℰ ∈ E_n(f))
14           if(c_i ∈ C_ℰ(ℰ))
15             C_rd = C_rd ∪ {(l_{i,j})};
16   return (⟨C_{M_rd}, 𝓕_rd, C_rd, 𝒢⟩);
```

**Fig. 4.** Reduction algorithm

```
1  function fwdCntx(⟨L, l^0, E, T⟩, F, l_0)
2    L^new = {l_0};
3    do
4      L^old = L^new;
5      L^new = L^new ∪ {l' ∈ L : ∃(l, e, l') ∈ T :
6                  l ∈ L^new ∧ effVal(e, F)};
7    while (L^old ≠ L^new);
8    E = E \ {e ∈ E : ∄(l, e, l') ∈ T :
9              l, l' ∈ L^new};
10   T = T \ {(l, e, l') ∈ T : e ∉ E};
11   L = L^new;
12   return ⟨L, l^0, E, T⟩;
13
14 function fwdFrgm(⟨S, s^0, A, T, Ann⟩, C_M)
15   S^new = {s^0};
16   do
17     S^old = S^new;
18     S^new = S^new ∪ {s' ∈ S : ∃(s, a, s') ∈ T :
19                 s ∈ S^new ∧ precVal(a, F, C_M)};
20   while (S^old ≠ S^new);
21   A = A \ {a ∈ A : ∄(s, a, s') ∈ T : s, s' ∈ S^new};
22   T = T \ {(s, a, s') ∈ T : a ∉ A};
23   S = S^new;
24   return ⟨S, s^0, A, T, Ann⟩;
25
26
27 function optimize(⟨C_M, F, l_C^0, G⟩)
28   F^new = F; C_M^new = C_M;
29   do
30     F^old = F^new; C_M^old = C_M^new;
31     F^new = ∅; C_M^new = ∅;
32     foreach(c ∈ C_M^old)
33       C_M^new = C_M^new ∪
34            {fwdCntx(c, F^old, l_C^0 ↓_c))};
35     foreach(f ∈ F^old)
36       f' = fwdFrgm(f, C_M^new);
37       if(!hasEffect(f'))
38         F^new = F^new ∪ {f'};
39   while(F^new ≠ F^old ∨ C_M^new ≠ C_M^old );
40   return (⟨C_M^new, F^new, l_C^0, G⟩);
```

**Fig. 5.** Optimization algorithm

**Step 1** (lines 32-34) is the reachability analysis of context properties ($l_C^0 \downarrow_c$ returns a state of context property $c$ for context configuration $l_C^0$ using set-based projection). We remark that context model $C_M$ is the reduced one take form algorithm in Fig.4. The core of this step is function `fwdCntx(`$c$`, `$\mathcal{F}$`, `$l_0$`)` (lines 1-12), which figures out the portion of the original context property $c$ that can be reached from the current state $l_0$ by executing fragments $\mathcal{F}$. It is implemented as a largest fixed point loop (lines 3-7), in which it is assumed that a context property can evolve only through transitions that can be triggered by available fragments (in lines 5-6 function `effVal(e,`$\mathcal{F}$`)` is used to identify if there exists at least one action among $\mathcal{F}$ that triggers $e$). All the irrelevant elements are removed from the original context property (lines 8-10);

**Step 2** (lines 35-38) is the reachability analysis for fragments exploiting `fwdFrgm` function (lines 13-23). Function `fwdFrgm(`$f$`, `$C$`)` figures out the portion of the original fragment $f$ that can be reached from its current state in compliance with context $C$. It is implemented as a largest fixed point loop (lines 16-20), in which it is assumed that a fragment can evolve only through transition that can be executed in the current context model without violating their preconditions (in lines 18-19 function `precVal(a,` $\mathcal{F}$`, `$C$`)` is used to identify if action $a$ belonging to some fragment among $\mathcal{F}$ can ever have its precondition satisfied in context model $C$). All the irrelevant elements are removed from the original fragment (lines 21-23). After `fwdFrgm` is called, we check (line 37) if the resulting fragment $f'$ obtained can produce any contextual affect in the current contextual situation (using function `hasEffect(`$f'$`)` we check if fragment $f$ still contains at least one action that is either abstract or can potentially produces some affect). If it is the case, $f'$ is added to the collection of fragments, if not, it is considered as useless (it by now means can change the context) and is filtered out.

We remark that any optimization in Step 1 may enable further optimization in Step 2 and vice versa. This is why the optimization process have to continue until a fixed point is reached.

### 4.2 Plan or Reuse

After that an adaptation problem has been optimized, the next step is to check if a solution for it has already been found in the past or not. Since the context $C$ and the goal $\mathcal{G}$ of an adaptation problem are expressed in terms of property instances, we have to abstract them in a way that they refer to context model. To this end we define an *abstract adaptation problem* as:

**Definition 10 (Abstract Adaptation Problem).** *An* abstract adaptation problem *is a tuple* $\xi_{abs} = \langle C_M, C_{abs}, \mathcal{F}, C, \mathcal{G}, \mathcal{G}_{abs}\rangle$*, where:*

- $C_M$*,* $\mathcal{F}$*,* $C, \mathcal{G}$ *are as Definition 9 ;*
- $C_{abs}$ *is a set of pairs of the form* $(i, l_{i,j})$ *where* $c_i$ *is a property model (type), while* $l_{i,j}$ *is the state of its instance;*
- $\mathcal{G}_{abs}$ *is a set of pairs of the form* $(i, l_{i,j})$ *indicating that a context property of type* $c_i$ *should reach the state* $l_{i,j}$*.*

We indicate with the symbol $\xi_{abs^\bullet}$ the triplet $\langle C_{abs}, \mathcal{F}, \mathcal{G}_{abs}\rangle$ belonging to a abstract problem $\xi_{abs}$.

```
1  function abs (C_M, F, C, G)
2    C_abs = ∅; G_abs = ∅;
3    foreach( l_{i,j} ∈ C )
4      C_abs = C_abs ∪ {(i, l_{i,j})};
5    foreach(l ∈ G)
6      foreach(l_{i,j} ∈ C )
7        if(l ↓ c_i ∈ l}
8          G_abs = G_abs ∪ {(i, l_{i,j})};
9    return ⟨C_abs, G_abs⟩;

10 function find(⟨C_M , C_abs , F , C ,
11                G , G_abs , Ψ⟩)
12   ξ_abs• = ⟨C_abs , F , G_abs⟩;
13   if( Ψ(ξ_abs•) == ⊥ )
14     solution = planner(C_M, F, C, G);
15     Ψ = Ψ[ξ_abs• , solution];
16     return solution;
17   else
18     return Ψ(ξ_abs•);

19 function grnd(f, C_M, C)
20   A' = ∅;
21   foreach(a ∈ A(f))
22     l = Ann(a);
23     foreach c_{i,j} ∈ C
24       if( l ↓ c_i ∈ l ∧ (j, l ↓ c_i) ∈ L_C ↓_{i,j} )
25         A' = A' ∪ {j, l ↓ c_i};
26   A(f)=A';
27   return f;
28
29 function reuse(C_M, F, C, G)
30   ⟨C_abs, G_abs⟩ = abs(C_M, F, C, G);
31   f = find(⟨C_M, C_abs, F, C, G, G_abs, Ψ⟩);
32   return grnd(f, C_M, C);
```

**Fig. 6.** Reuse Algorithm

Figure 6 shows the reuse algorithm (lines 29-32). It is composed by three steps: problem abstraction, finding a general solution and then grounding the solution to the actual runtime context. Once the abstract adaptation problem has been computed (function *abs* of Figure 6), then the function *find* checks if a solution for it has already been calculated in the past. To this end, we use a lookup function $\Psi$ from abstract problems to fragments. If an abstract problem does not belong to the domain[1] of $\Psi$ (e.g. the abstract problem is a new one) then the solution is calculated by invoking the adaptation engine and then $\Psi$ is augmented with the new solution (line 15 of Figure 6). We define $\Psi[\xi_{abs^\bullet}, f](\xi^1_{abs^\bullet})$ as:

$$\Psi[\xi_{abs}, f](\xi^1_{abs^\bullet}) = \begin{cases} \Psi(\xi^1_{abs^\bullet}) & \text{if } \xi^1_{abs^\bullet} \neq \xi_{abs^\bullet} \\ f & \text{otherwise} \end{cases}$$

[1] The function returns $\perp$ if a problem is not in its domain.

Solutions returned by the function *find* (and by the planner) are fragments, whose annotations are on context models and not on instances. We need then to ground the found solution to the level of the process that rose the adaptation, in terms of context property instances used by the process itself. To this end the function $grnd$ that substitutes all the (context model) annotations of a fragment with runtime context state annotations.

## 5 Experiments and Results

The proposed framework has been implemented as an extension of the ASTRO-CAptEvo framework [14] and evaluated on a real world scenario based on the operation of car logistics in the Bremen sea port [3]. We show that it is effective in reducing the number of requested adaptations: while the situation where new adaptation are not needed is never reached (thus witnessing the need of dynamic adaptation), the number of such new adaptations decreases over time (thus making reuse more and more efficient). We also show that the approach seamlessly accommodates situations where previous adaptations are not valid any-more, e.g., due to changes in the requirements or in the available fragments. 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.

During the experiments[2], we collected the number of adaptation cases and the time to generate an adaptation solution for each case. We remark that preliminary optimization steps described in Section 4 require significantly less time than planning itself and thus do not contribute dramatically to the overall solution search time.

In the first experiment we measure the effectiveness of the proposed optimization approach, i.e., of the possibility to reuse previous solutions. We conducted four simulations in different configurations, each one differing from the others in terms of frequency of ship arrivals, car damages and delivery orders. The chart in Figure 7 plots on the X-axis the number of adaptation cases (the first 2500 for each simulation) and on the Y-axis the number of unique adaptation problems (i.e., number of adaptations actually computed in the presence of our reuse mechanism). The dashed curve represents the old (naive) approach without the reuse mechanism. Indeed the curve is linear for all the experiments, meaning that for each adaptation problem that comes a solution is calculated from scratch. The other curves correspond to the four simulations with the reuse. In the worst case, we have $118$ unique adaptation problems out of $2500$ adaptation cases, that is in $94.3\%$ of the cases we are able to reuse existing solutions. On average for the four simulations, the reuse rate was $95.5\%$. In terms of time (not shown in the figure), the computation of all 2500 adaptations solutions *without* adaptation problem optimization required $4821$ seconds, while the optimization mechanism reduced this time to $306$ seconds ($6.34\%$ of the non-optimized time). Figure 7 also shows that new adaptation problems (where reuse was not possible) keep emerging through all the simulation, even though their rate reduced over time. This indicates the importance of on-the-fly adaptation as opposed to static predefined solutions. Indeed, after the initial phase (first $1000$ adaptation cases) new adaptation problems emerge quite rarely, but

[2] All the experiments were executed on a Linux machine with 4-core Intel i7 CPU running at 2.3GHz with 16GB of memory.

even after 2500 cases they do not disappear completely (i.e., the complete reuse was not possible).

In order to see how the new framework can deal with highly dynamic systems, we measure the impact of various dynamic factors (in particular, changes in requirements and fragments) on its operation and performance. More precisely, we compare the normal simulation of the system with the one where two dynamic factors are enabled. These factors are (i) the introduction of a new type of entity (in our case, the luxury car) and (ii) the change in a single fragment (in our case, we change the procedure for storing cars in the consignment area). In the first case, all the adaptation solutions involving luxury cars have to be generated anew (no reuse is possible). In the second case, all the adaptation cases involving the changed fragment have also to be regenerated. Both dynamic changes result in increase of a number of unique adaptation cases (and in additional workload for the composition engine). To reflect this in figures, through the simulation we measure the number of calls to the composition engine for the last 40 adaptation cases (thus reflecting the number of new adaptation problems for which reuse is not possible). These data are shown in Figure 8. The solid red line reflect the rate of new adaptation problems attributed to the normal run of the system, while the dashed blue one reflects only those new adaptation cases that were triggered by dynamic factors. From the red line it can be seen that the high rate of new adaptation cases is common for the initial phase of the simulation (first $800 - 1000$ adaptation cases) but even after 2000 cases into the simulation it is still non-zero. From the blue

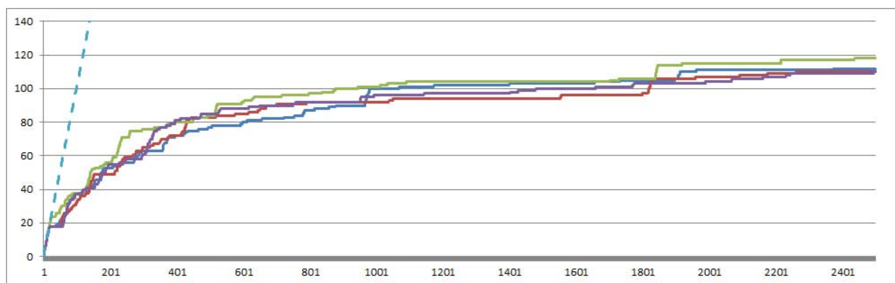


**Fig. 7.** New adaptation problems versus total number of adaptation needs

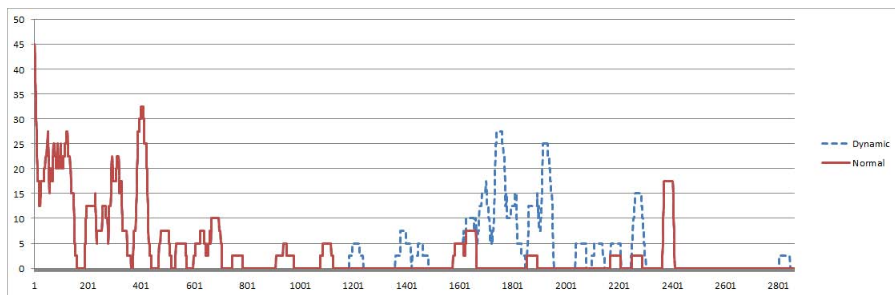


**Fig. 8.** Impact of dynamic factors on the number of new adaptation problems

line, it is evident that new adaptation problems start to emerge after 1200 cases (which actually corresponds to the introduction of a new entity type) and has a prominent peak around 1700-th case (which corresponds to the change in the fragment). It is worth to note that introducing a new entity brings immediately just a small amount of new adaptation problems (three peaks between the 1200-th and 1600-th problem) while the change in the fragment that is involved in many various adaptations produces considerable immediate surge. Again, after having computed the new problems brought by the changes, the system behaves as the normal execution (e.g. the ratio of new problems keeps decreasing but never reaches 0).

## 6 Related Work and Conclusion

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 [15], and optimizing QoS of service-based applications [11,17,19], or for satisfying some application constraints [8,16]. Repairing strategies could be specified by means of policies to manage the dynamism of the execution environment [1,6]. The aim of the strategies proposed by the aforementioned approaches range from service selection to rebinding and application reconfiguration [12,18]. These are interesting features, but cannot deal with complex and dynamic service-based systems where context-awareness and adaptivity are key characteristics.

In this paper, we have proposed an *on-the-fly* adaptation approach where adaptation activities are not explicitly represented at design time but are discovered and managed on-the-fly considering all aspects of the execution environment (current context, available process fragments, etc.). This means that if an adaptation solution exists, our approach will find it automatically, without involving off-line activities. Moreover, our approach is able to reduce the complexity of each adaptation problem by minimizing the search space according to the specific execution context, and reuse adaptation solutions by learning from past executions.

The type of systems that our approach can deal with have the characteristic to be dynamic in terms of number of entities involved and number of adaptation problems to consider. At the same time each adaptation problem involves a potentially very large set of available fragments. Thanks to the two steps defined in our approach: *problem reduction* and *solution reuse* we are able to solve such a large number of adaptation problems on-the-fly and in a reasonable execution time.

As future work, we want to extend the framework in a such a way that it will be also *user-centric*. In user-centric systems [13], services are intended to be consumed directly by the user (e.g., personal agenda, on-line flight booking, etc.). While in our approach process fragments are orchestrated in order to accomplish a specific business task, user centric systems should allow the user to decide and control which tasks are executed and how. This requires to extend our approach with the ability not only to automatically compose and adapt different, often unrelated, fragments on the fly, but also to generate a flexible interaction protocol that allows the user to control and coordinate the composition execution.

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

## References

[1] Baresi, L., Guinea, S., Pasquale, L.: Self-healing BPEL processes with Dynamo and the JBoss rule engine. In: Proc. of ESSPE 2007, pp. 11–20. ACM (2007)
[2] Bertoli, P., Pistore, M., Traverso, P.: Automated composition of web services via planning in asynchronous domains. Artif. Intell. 174(3-4), 316–361 (2010)
[3] Böse, F., Piotrowski, J.: 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] Bucchiarone, A., Marconi, A., Pistore, M., Raik, H.: Dynamic Adaptation of Fragment-based and Context-aware Business Processes. In: Proc. of ICWS 2012, pp. 33–41 (2012)
[5] Bucchiarone, A., Antares Mezzina, C., Pistore, M.: Captlang: a language for context-aware and adaptable business processes. In: Proc. of VaMoS 2013, pp. 12:1–12:5. ACM (2013)
[6] Colombo, M., Di Nitto, E., Mauri, M.: SCENE: A service composition execution environment supporting dynamic changes disciplined through rules. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 191–202. Springer, Heidelberg (2006)
[7] Dal Lago, U., Pistore, M., Traverso, P.: Planning with a Language for Extended Goals. In: Proc. of AAAI 2002 (2002)
[8] de Leoni, M.: Adaptive Process Management in Highly Dynamic and Pervasive Scenarios. In: Proc. of YR-SOC, pp. 83–97 (2009)
[9] Eberle, H., Unger, T., Leymann, F.: Process fragments. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2009, Part I. LNCS, vol. 5870, pp. 398–405. Springer, Heidelberg (2009)
[10] Marconi, A., Pistore, M., Traverso, P.: Automated Composition of Web Services: the ASTRO Approach. IEEE Data Eng. Bull. 31(3), 23–26 (2008)
[11] Mirandola, R., Potena, P.: A qos-based framework for the adaptation of service-based systems. Scalable Computing: Practice and Experience 12(1), 63–78 (2011)
[12] Pfeffer, H., Linner, D., Steglich, S.: Dynamic adaptation of workflow based service compositions. In: Huang, D.-S., Wunsch II, D.C., Levine, D.S., Jo, K.-H. (eds.) ICIC 2008. LNCS, vol. 5226, pp. 763–774. Springer, Heidelberg (2008)
[13] Pistore, M., Traverso, P., Paolucci, M., Wagner, M.: From software services to a future internet of services. In: Proc. of FIA 2009, pp. 183–192 (2009)
[14] Raik, H., Bucchiarone, A., Khurshid, N., Marconi, A., Pistore, M.: Astro-captevo: Dynamic context-aware adaptation for service-based systems. In: Proc. of SERVICES 2012, pp. 385–392 (2012)
[15] Spanoudakis, G., Zisman, A., Kozlenkov, A.: A service discovery framework for service centric systems. In: Proc. of IEEE SCC 2005, pp. 251–259 (2005)
[16] Verma, K., Gomadam, K., Sheth, A.P., Miller, J.A., Wu, Z.: The METEOR-S approach for configuring and executing dynamic web processes. Technical report, University of Georgia, Athens (2005)
[17] Wang, C., Pazat, J.L.: A two-phase online prediction approach for accurate and timely adaptation decision. In: Proc. of SCC 2012, pp. 218–225. IEEE Computer Society (2012)
[18] Yan, Y., Poizat, P., Zhao, L.: Self-adaptive service composition through graphplan repair. In: Proc. of ICWS 2010, pp. 624–627 (2010)
[19] Zhai, Y., Zhang, J., Lin, K.: Soa middleware support for service process reconfiguration with end-to-end qos constraints. In: Proc. of ICWS 2009, pp. 815–822 (2009)