

RESEARCH

Open Access



# A context-aware framework for dynamic composition of process fragments in the internet of services

Antonio Bucchiarone\* , Annapaola Marconi, Marco Pistore and Heorhi Raik

## Abstract

In the last decade, many approaches to automated service composition have been proposed. However, most of them do not fully exploit the opportunities offered by the Internet of Services (IoS). In this article, we focus on the dynamicity of the execution environment, that is, any change occurring at run-time that might affect the system, such as changes in service availability, service behavior, or characteristics of the execution context. We indicate that any IoS-based application strongly requires a composition framework that supports for the automation of all the phases of the composition life cycle, from requirements derivation, to synthesis, deployment and execution. Our solution to this ambitious problem is an AI planning-based composition framework that features abstract composition requirements and context-awareness. In the proposed approach most human-dependent tasks can be accomplished at design time and the few human intervention required at run time do not affect the system execution. To demonstrate our approach in action and evaluate it, we exploit the ASTRO-CaptEvo framework, simulating the operation of a fully automated IoS-based car logistics scenario in the Bremerhaven harbor.

**Keywords:** Internet of services, Dynamic service composition, Process fragment, Context-aware, AI planning

## 1 Introduction

Service composition is one of the cornerstone technologies within service-oriented computing. It consists in reusing existing services as building blocks for new services (applications) with higher-level functionality. Service composition allows for extremely rapid software development and high re-usability of development results.

Despite all the advantages service composition brings to software engineers, when performed manually, it is still a very complex, time-consuming and error-prone task. The point is that composition requirements and service specifications usually contain numerous easy-to-miss technical details that have to be properly reflected in the composition. This becomes critical when composition has to be exploited in *complex and dynamic* application domains, requiring frequent revision of providers (i.e., component services), changes in existing offered functionalities (i.e. component services behavior), and adjustment of business policies and objectives (i.e., composition requirements).

This is the case for IoS-based application domains, where the execution environment is so dynamic, that service composition is considered to be a kind of every-minute routine activity.

We can mention at least two important examples of such IoS-based domains. The first one are *pervasive systems*, which are mobile systems operating in close connection with their context. Once service composition is exploited in this setting, it has to be flexibly and quickly adapted to the rapidly changing environment. For instance, let us imagine there is a car that has to regularly perform some activity implemented through composition of surrounding near-field communication services (e.g., car parking assisted by various parking services). Depending on the current surrounding environment (i.e., on the set of available services and on the context state) the solution composition, though targeting conceptually the same objective, will always be different. If the procedure is repeated frequently at different locations, we have to produce new compositions again and again. Another example are *user-centric systems*, whose operation evolves around the needs and constraints of a specific user. For instance, it could

\*Correspondence: [bucchiarone@fbk.eu](mailto:bucchiarone@fbk.eu)  
Fondazione Bruno Kessler, Via Sommarive, 18, 38123 Trento, Italy

be a mobile application that allows the user to integrate (compose) multiple mobile services (local phone services, Internet services, near-field communication services, etc..) and execute them consistently. In this case, the choice of services and composition objectives are determined by user's environment and personal preferences/constraints/goals.

It is quite clear that predefined solutions are not going to work in these IoS-based systems. Indeed, each composition heavily depends on the run-time parameters of the execution environment, namely, current execution context, set of available services, concrete user's needs, etc. Since these parameters are not predictable at design time (sometimes we may not even know which services will be available at the time of composition), it is impossible to produce reliable solutions a priori. In order to address this problem we need a *dynamic composition framework* that would automate the whole service composition life-cycle, from requirements derivation, to composition generation, to deployment and execution [1, 2].

Another important aspect that directly follows from system dynamicity is *context-aware composition and execution*. Services are often closely connected to the context in which they are executed (e.g., a parking web service execution may depend on the type of a car to be parked, on space availability, weather, etc.). In turn, the context tends to be volatile, i.e. exogenous events may change the context state and thus affect composition execution. In this setting, to be able to produce compositions that are consistent with the surrounding context it is important to have a context-aware service model that reflects contextual characteristics of services. This information has to be taken into account in service composition, thus enabling more robust solutions. It is worth to notice that the problem of context-awareness of service composition has not yet received enough attention from the scientific community and only very few works are currently available on the topic [3–5]. Many existing approaches to service composition suffer from over simplification, both for what concerns the service model (e.g., services are often considered as atomic synchronous operations) and composition requirements (e.g., requirements languages do not reflect real-world composition needs). As a result, these solutions can only be applied to very limited set of composition problems. At the same time, a few approaches demonstrate more maturity in addressing these aspects.

In this article, for the first time, we present a comprehensive framework, and its implementation, for automated service composition that is specifically designed to be used in dynamic execution environments and allows for context-aware service composition and execution.

In very general terms, the idea consists in organizing the composition life cycle in a such a way that most human activities can be accomplished at design time.

As a consequence, run-time composition management, from the derivation of composition requirements to the composition synthesis to the deployment of executable processes, is completely automated. Moreover, whenever human involvement is required at run-time (e.g., plugging-in a new service into the system), the change is automatically dealt by the framework, without affecting the system execution. In addition to that, our composition framework features explicit context model that is used to express various contextual characteristics of services. Later, these characteristics can be taken into account in automated reasoning so that context-aware service compositions are produced. The approach exploits AI planning techniques that can deal with realistic service models (asynchronous, stateful and nondeterministic services) and allows for rich control- and data-flow requirements<sup>1</sup>. This makes it powerful enough to be used in real service-based systems, including those mentioned above. The demonstration is given also by its usage, as a core component, in different previous works to support the incremental refinement and adaptation of context-aware service based systems [6–8].

In the article, as composable components we use *process fragments* (or simply fragments). Fragments [9] are a way to represent reusable process knowledge in service compositions and encode elementary subprocesses that can be used as constructing blocks for more complex processes. Process fragments are also a very effective mean to model stateful and asynchronous services [10].

The rest of the article is structured as follows. In Section 2, we present the Car Logistics scenario used as a reference throughout the article and explain the challenges it poses to service composition. In Section 3, we present our composition framework, formally defining all the concepts and models used for context-aware dynamic service composition and show how the composition problem is solved through AI planning techniques. The experimental validation of the approach can be found in Section 4 where we have used the implementation of the Car Logistics scenario to demonstrate our approach in action and evaluate its performance and scalability. Finally, we conclude the article with related work survey (Section 5) and conclusions (Section 7).

## 2 Motivating scenario and research challenges

In this Section we introduce the car logistics scenario that helps understanding the problem we want to solve and the research challenges that it poses.

### 2.1 Car logistics scenario

The motivating scenario (later referred to as Car Logistics Scenario or CLS) is inspired by the operation of the seaport of Bremerhaven (Bremen, Germany) [11]. Every year this port receives around 2 million cars transported

by ships in order to further deliver them to retailers. The delivery process for each car (see Fig. 1) includes a number of steps to be accomplished. Cars arrive by ship and each ship is able to approach and leave a gate interacting with the landing manager, which is in charge of coordinating and controlling the landing procedure for all the ships in the port. Then cars are unloaded and unpacked at one of the terminals. Once a car is unpacked, it has to be moved to one of the storage areas: the chosen area depends on the car type/brand and on the availability of parking spaces; different storage areas have different parking procedures that need to be followed. The 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, the car is treated at dedicated treatment areas (e.g., washing, painting, equipment) according to the details of the order and the car brand/model. 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. It is important that every step in the chain is customizable according to the car brand, model, retailer requirements, etc. This means different cars may utilize different procedures to accomplish the same task.

These procedures are partially automated and exploit digital services as well as sensors and smart manufacturing equipment [11]. We tried to move this scenario a little further to fully capture the opportunities offered by paradigms such as IoS and IoT, yet keeping its current needs, regulations and practices. Our goal was to develop a service-based system that supports the synergistic cooperation of the numerous actors (i. e., cars, ships, trucks, treatment areas, parking facilities, drivers, etc.) allowing them to follow their current procedures and business policies. The CLS scenario presents several example of dinamicity that we list in the following.

- *Customization.* The system should consider the customization of each car procedure. It means that different brands and models of cars in a similar but customizable way according to the specific order and context (e.g., facility in which they are treated).
- *Openness.* The system should be able to easily integrate new actors and services at run-time. This

happens, for example, when new car models, having specific requirements and procedures, is handled at the seaport, when a new truck/ship company comes into play, as well as whenever there is a new functionality provided by the sensors and machineries used in the different port facilities.

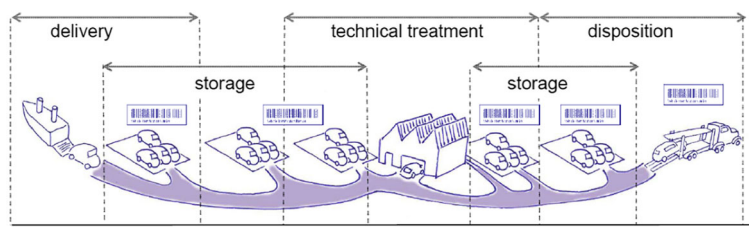
- *Flexibility.* The system needs to flexibly deal with changes in the procedures and business policies of all the actors such as ships (e.g., changes in approaching procedure), trucks (e.g., changes in delivery procedure), port facilities (e.g., update in a parking service or in a certain procedure supported by a machinery in a treatment facility).
- *Context-awareness.* The system needs to deal with situations where contextual changes invalidate some choices made before. For example, if a car books a space at some storage area and upon arrival realizes that the facility is not available, it needs to rearrange the things so that it can be stored at a different facility.

### 2.2 Research challenges

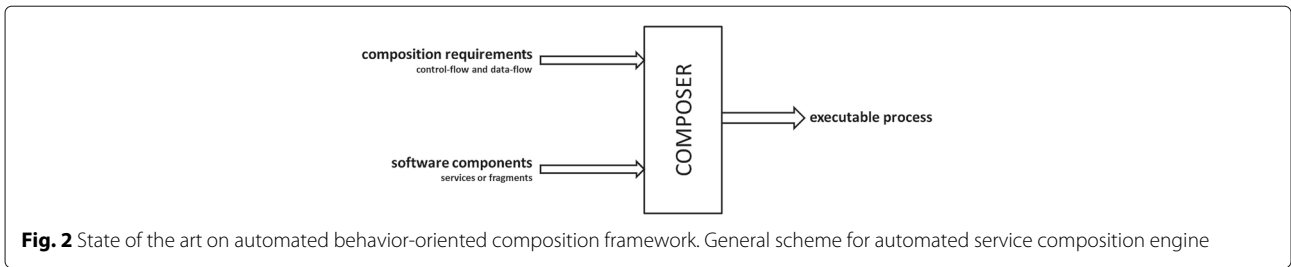
In the previous section, we gave an example of a system where dynamic composition techniques are strongly required. As a result, it is hardly affordable to involve human experts to participate in resolving composition problems on-line. This essentially means that all the composition steps, from abstract requirements' specification, to service discovery and composition, to service deployment, must be automated.

This challenge is hardly coped with by the existing composition techniques. In general, every automated composition engine follows the scheme shown in Fig. 2. As input, it takes specifications of *software components* and *composition requirements* and, as output, it produces an executable process.

The major problem of most existing composition tools is that they assume that the set of services to be composed is always known to the designer at the moment of specifying composition requirements. Consequently, it is assumed that composition requirements may include implementation-specific details of services that they are supposed to be used with. As such, requirements become linked to particular service implementation and cannot



**Fig. 1** Process chain of the car logistics scenario. Delivery process of a car with its steps to be accomplished



further be used with conceptually similar services that are implemented differently (e.g., when we would like to use a new ticket booking service in place of the one for which the requirements were originally created). In dynamic systems this causes an important problem: while the list of available service is dynamic and can only be discovered at run time, the composition requirements are fixed at design time. Consequently, we need to understand *how to create a composition architecture that allows design-time requirements to be consistently used with an arbitrary set of services discovered at run time with no human intervention* (Challenge 1).

Another problem concerns *context-awareness*. In several cases, process execution is tightly connected to the context. When numerous services are available, *it is critical to select correct services and produce a composition that is valid for the given context model and for the current state of this model* (Challenge 2).

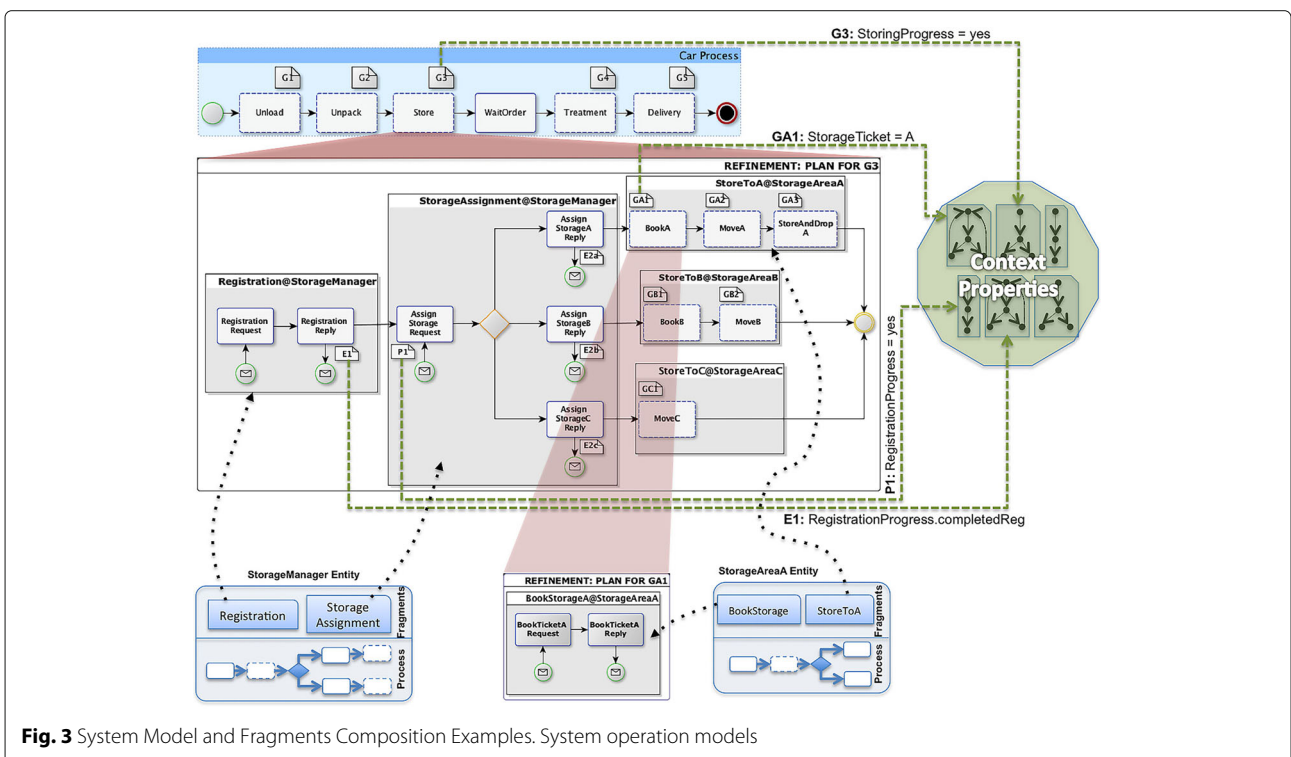
Finally, we need to recall that composable components may be quite complex and may feature 1) *statefulness* (complex protocol), 2) *non-determinism* and 3) *asynchronous* communication. As a result, *the composition architecture has to rely on a service model that reflects these properties of real services* (Challenge 3).

### 3 Dynamic service composition framework

In this section we present our framework for modeling dynamic context-aware systems such as the CLS scenario described in Section 2.

#### 3.1 System model

The system operation is modeled through a set of *entities* (e.g., cars, ships, storage managers, etc..), each specifying its behavior through a *process*, as depicted in Fig. 3. Unlike traditional system specifications, where processes are static descriptions of the expected run-time operation,



our approach allows to define dynamic processes that are refined at run time according to the features offered by the system.

Processes are designed in Adaptable Pervasive Flows Language, (APFL) [12, 13], an extension of traditional workflow languages (e.g., BPEL [14]<sup>2</sup>), which makes them suitable for adaptation and execution in dynamic environments. Unlike traditional processes, where their behaviors are completely specified, our approach allows the partial specification through *abstract activities* that can be specialized at run-time according to the services offered by the other entities in the system.

Abstract activities (e.g., *Store* of the *Car* entity process in Fig. 3) corresponds to tasks that are hard to implement at design time, since they strongly depend on the concrete run-time state (context configuration, availability of services provided by other entities etc.). In our approach, each abstract activity is dynamically replaced (refined) with a fragment composition realizing the corresponding task. As a result, a design-time specification of a process can be quite abstract, with many concrete details being refined only at run time, when the actual execution context is clear.

In addition to the classical workflow language constructs (e.g., input, output, data manipulation activities, complex control flow constructs), APFL adds 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.

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 processes.

For instance, within the CLS, whenever a *car* must be stored, it discovers the fragments provided by the *Storage Manager* and by the associated *Storage Area* (A, B, or C in Fig. 3). These fragments model the harbor-specific procedures and regulations that the car should execute for the storing. Different fragments may be provided by the storage manager and by the different storage areas to be used by certain types of car.

In our framework, we use a unified model for both fragments and fragment-based processes, and uniformly use the term of *fragment* for both of them. We model fragments as labelled transition systems (LTS) (as depicted in Figs. 5 and 6) where transitions are labelled with two different types of actions: *controllable* and *uncontrollable*. Controllable actions are used to model process activities that can be triggered by the process itself (e. g., variable

assignment, message send). Uncontrollable actions model activities whose execution depends on external actors (e. g., message receive, event notification). The distinction between controllable and uncontrollable actions is crucial for proper handling of the asynchronicity of fragment behaviour in composition. Here and later in the text we denote with ‘!’ and ‘?’ controllable and uncontrollable fragment actions respectively. The fragment LTS is formally defined as follows:

**Definition 1** (Fragment) *A fragment is a deterministic state transition system  $f = \langle S, s^0, I, O, R \rangle$ , where*

- *S is the set of states and  $s^0 \subseteq S$  is the initial state;*
- *I and O are sets of controllable and uncontrollable actions such that  $I \cap O = \emptyset$ ;*
- *$R \subseteq S \times \{I \cup O\} \times S$  is a transition relation.*

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 process. This kind of modeling extremely simplifies the specification of 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).

The aim of context properties is to model those aspects of the context that are relevant for dynamic fragments’ composition. Their intent is not to provide a comprehensive and detailed representation of the properties and state of the execution environment. Rather, they are an abstraction of the context, capturing only key domain concepts (e.g., car, ship, storage area) and their evolution. These information are used to reason on how fragments execution (preconditions/effects) is related to and affects the context state.

Context property behaviour is described by a labelled transition system that contains all possible states of the context property and transitions between them<sup>3</sup>. Each transition is labeled with a context event. Formally:

**Definition 2** (Context Property) A context property is a state transition system  $p = \langle L, l^0, E, T \rangle$ , where:

- $L$  is a set of context states and  $l^0 \in L$  is the initial state;
- $E$  is a set of context property events;
- $T \subseteq L \times E \times L$  is a transition relation.

Examples of context properties (of a car) are shown in Fig. 4. It includes a complex Location property reflecting current car location and, two of progress tracking properties (RegistrationProgress, StoringProgress).

Considering a context model containing more than one context property, we require that such context properties feature mutually disjoint sets of context events, so that evolutions of different context properties within the same context do not correlate explicitly. Formally, context is defined as follows:

**Definition 3** (Context) A context is a set of context properties  $C = \{p_1, p_2, \dots, p_n\}$  such that if  $p_i = \langle L_i, l^0_i, E_i, T_i \rangle$  for all  $i \in [1, n]$  then for any two constituent context properties  $p_i, p_j \in C$  sets of events do not intersect (i.e.,  $E_i \cap E_j = \emptyset$ ). The set of all context states is defined as  $L_C = \prod_{i=1}^n L_i$  and the initial context state is  $l^0_C = (l^0_1, l^0_2, \dots, l^0_n)$ . We also denote a set of all context events as  $E_C = \bigcup_{i=1}^n E_i$ .

In order to be able to succinctly specify groups of context states we use context formulas which are disjunctions of conjunctions over states of context properties:

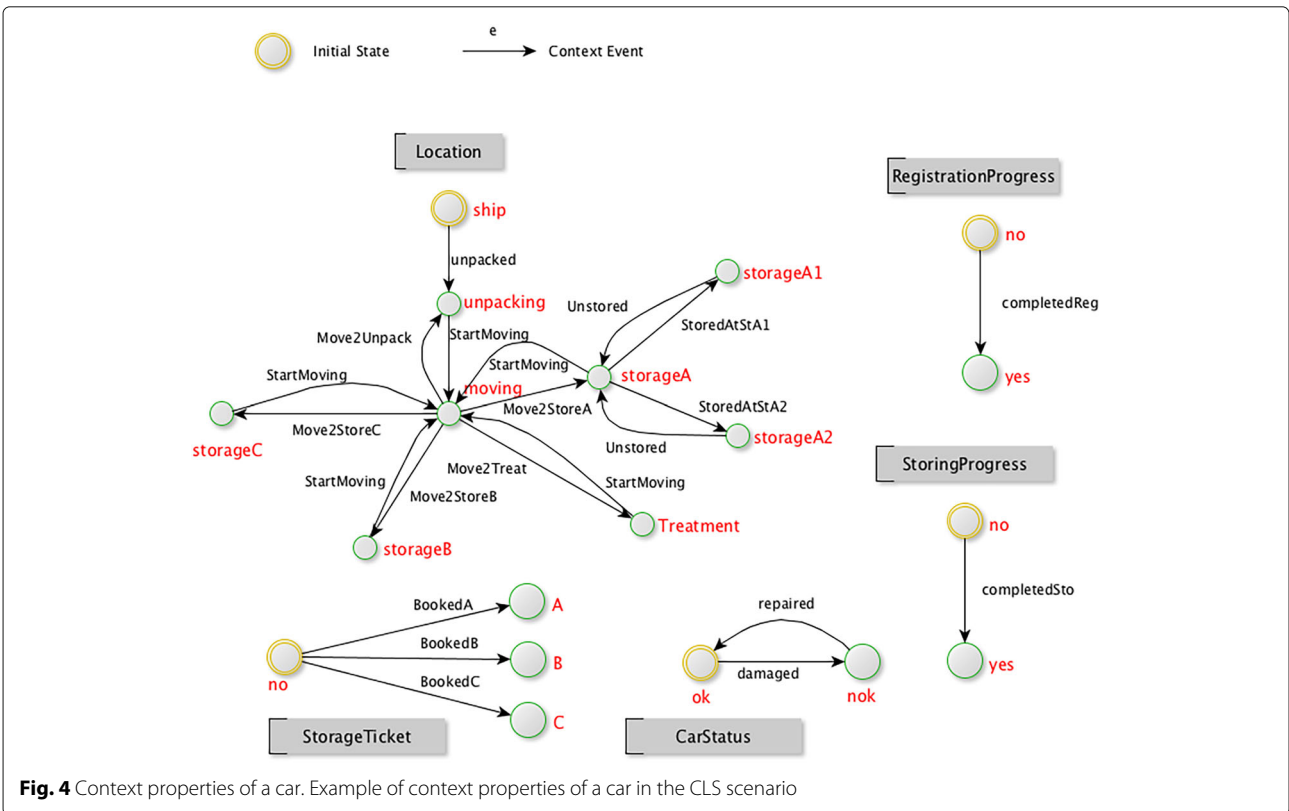
**Definition 4** (Context Formula) Let  $C = \{p_1, p_2, \dots, p_n\}$  be a context such that  $p_k = \langle L_k, l^0_k, E_k, T_k \rangle$  for all  $k \in [1, n]$ . A state formula for  $C$  is a propositional formula  $\bigvee_i \bigwedge_j l_{ij}$ ,

where  $l_{ij} \in \bigcup_{k=1}^n L_k$  and for any two constituent context properties  $p_i, p_j \in C$  sets of states do not intersect (i.e.,  $L_i \cap L_j = \emptyset$ ).

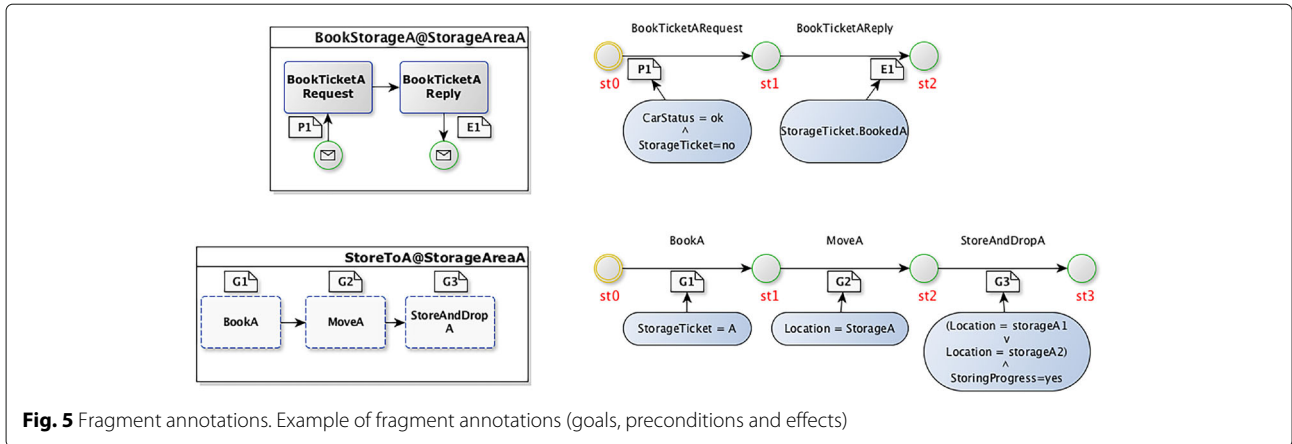
The space of all context formulas of context  $C$  is denoted as  $R_C$ .

In order to make fragments context-aware, we introduce context annotations of actions in fragments. Annotations intensively exploit the notion of context formulas, and can be of three types:

- action precondition is a context formula indicating in which context states action execution is allowed (e.g., P1 in Fig. 5);
- action effect is a set of context events that are triggered as a result of action execution (e.g., E1 in Fig. 5);



**Fig. 4** Context properties of a car. Example of context properties of a car in the CLS scenario



**Fig. 5** Fragment annotations. Example of fragment annotations (goals, preconditions and effects)

- action goal is a context formula specifying the condition that must hold after the action is executed (e.g., G1, G2 and G3 in Fig. 5).

Any process action can be annotated with a precondition. Since both effects and goals express the contextual intention of an action, we explicitly require that an action can be annotated either with a goal (if it is an abstract activity) or with an effect (for all other activities). Formally, *fragment annotation* is defined as follows:

**Definition 5** (Fragment Annotation) Let  $f = \langle S, s^0, I, O, R \rangle$  be a fragment and let  $C$  be a context. An annotation of fragment  $f$  over context  $C$  is a tuple  $\omega = \langle P, E, G \rangle$ , where:

- $P : \{I \cup O\} \rightarrow R_C$  is the precondition labeling function;
- $E : \{I \cup O\} \rightarrow E_C$  is the effect labeling function.
- Any action effect  $E(a)$  may contain no more than one event per context property; For any context property  $p = \langle L, I^0, E, T \rangle \in C$  the following holds:  $\nexists e_1, e_2 \in E(a) : e_1, e_2 \in E$ .
- If  $E(a) \neq \emptyset$  then  $G(a) = \emptyset$  (i.e., an action can be annotated either with a goal or with an effect);
- $G : \{I \cup O\} \rightarrow R_C$  is the goal labeling function, such that  $G(a) \neq \emptyset$  only if  $E(a) = \emptyset$  (i.e., an action can be annotated either with a goal or with an effect).

In Fig. 5 we show two fragments (e.g., *BookStorageA* and *StoreToA*), both provided by *Storage Area A* entity. The examples include all types of annotations. The annotations, in turn, are related to the context property examples in Fig. 4. For instance, *BookStorageA* includes two activities (request and reply). The request activity is annotated with a precondition that guards that ticket booking is executed only in the absence of a ticket. The effect of the reply activity indicates that this fragment eventually brings a ticket object to state A, thus providing ticket booking. *StoreToA* fragment specifies the

storing procedure for storage area A. This includes three abstract activities annotated with goals in terms of context model.

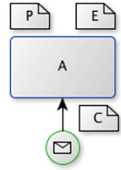
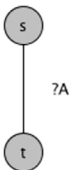
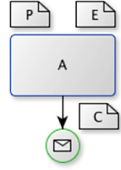
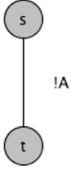
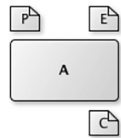
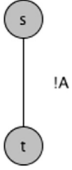

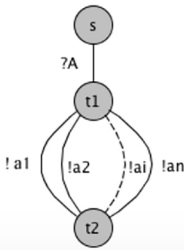
In the following we present the synopsis of our APFL language with annotations and the details of how an annotated APFL process can be transformed into an annotated LTS as defined in Def. 5. Since both fragments and processes are defined in the same language, the translations below are valid for both of them. In Fig. 6 the translation for basic activities is shown. Specifically, SEND and CONCRETE are represented with a single controllable transition, while RECEIVE is a single uncontrollable transition.

ABSTRACT activities are way more complex since at the moment of creating a new process they are not refined to a concrete process and the only thing we know about them is their abstract goals. We treat an abstract activity as a "black box" that performs a task as defined by its goal. In this regard, an abstract activity combines the properties of controllable and uncontrollable actions. On the one hand, the initiation of an abstract activity is controllable (within a process we can decide if to execute it and when). On the other hand, it is not possible to predict a priori the terminal context configuration. In LTS, such behaviour can be modeled as a controllable action followed by a number of uncontrollable actions corresponding to all possible terminal context states. We actually reduce the number of terminal states to the number of conjunctive clauses in the goal formula<sup>4</sup>.

### 3.2 Fragment composition model

The central idea of our framework for fragment composition can be captured from Fig. 7. The explicit model of the execution context in the center of the figure, is a collection of *context properties*.

In our approach, the context model is specifically used to reason on how a certain objective can be achieved through fragment execution, rather than for process modeling purposes.

APFL Activity	LTS	Annotation
<p>receive</p> 		$P(A) = P$ $E(A) = E$ $C(A) = C$
<p>send</p> 		$P(A) = P$ $E(A) = E$ $C(A) = C$
<p>concrete</p> 		$P(A) = P$ $E(A) = E$ $C(A) = C$
<p>abstract</p> <p><math>G = g_i \vee g_j \vee \dots \vee g_n \forall i \in \{1, \dots, n\}; g_i = \Lambda_j l_j</math></p> 		$G(a_i) = g_i$

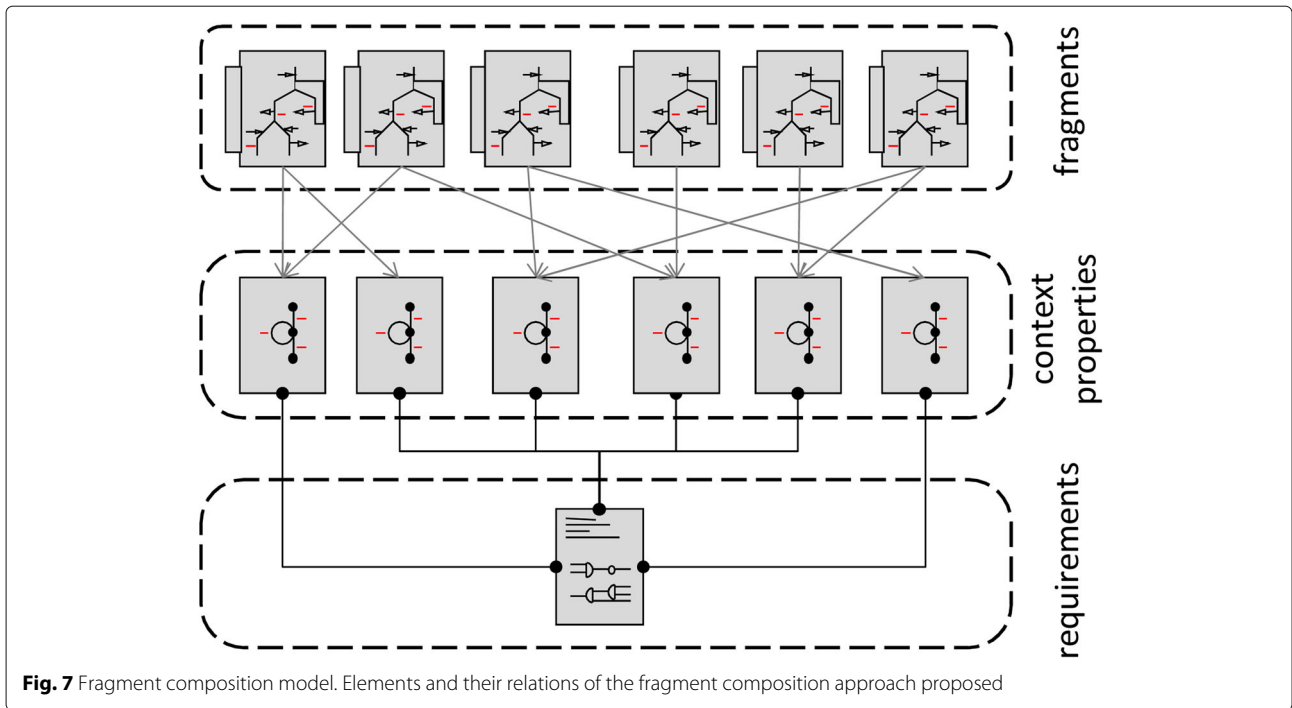
**Fig. 6** Translation of basic APFL activities into LTSs. Synopsis of our APFL language with annotations and the details of how an annotated APFL process can be transformed into an annotated LTS

To link fragments to context properties, we annotate the former with context-related information (as Defined in Def. 5 and depicted in Fig. 8). In this way, we explicitly connect the execution of fragment activities to states and transitions of context properties. In particular, a fragment activity may be annotated with *effect* and *precondition*. The effect shows which events this activity triggers once executed, while the precondition shows in which contextual states the execution of the activity is allowed.

Moreover, every abstract activity has a *goal* associated. Such goal specifies a context state(s) to be reached as a result of activity refinement and execution (e.g., if the *StorageTicket* is initially in state *no*, the goal of *BookA* activity in Fig. 3 may be to have this context property in state *A*).

One of the most important aspects of our approach is that *composition requirements* are expressed over context model, and not over fragments as done in most existing composition techniques. The core idea of our fragment composition model is that while fragment execution is closely related to context evolution, the modelling of the latter is solely determined by the application domain and does not depend on particular fragment implementations. As such, *by expressing composition requirements on the level of the context model on the one side, and by annotating fragments with context information on the other side, we create a composition framework in which composition requirements, though detached from fragment implementations, can always be automatically grounded on them.*





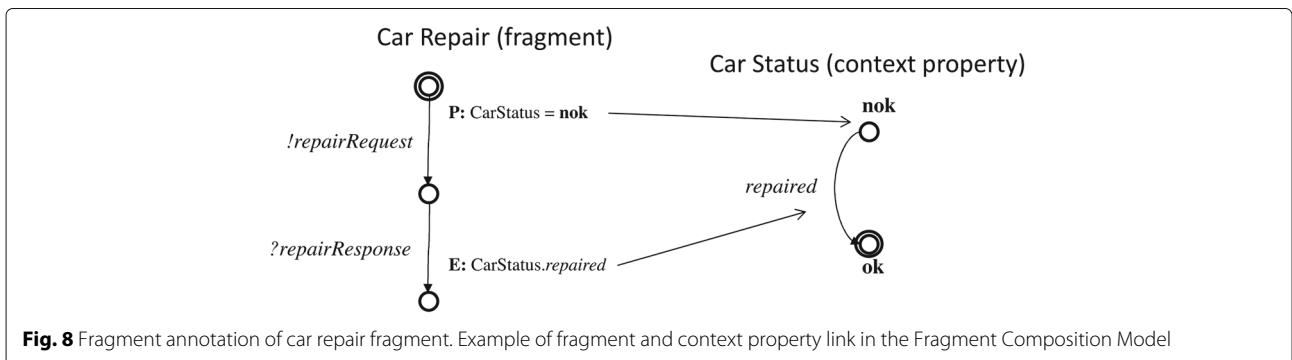
The ability to specify composition requirements on context properties makes it possible to efficiently use our approach in extremely dynamic environments, where both the set of available fragments and the execution context are constantly changing. At design time, the requirements are defined only conceptually (with no adherence to any particular set of fragments). Despite we do not know a priori the set of fragment, through annotation-based grounding, the conceptual requirements can always be restated for the actual (dynamically discovered) set of fragments. As such, we automate one of the most critical steps in service composition: run-time derivation of composition requirements.

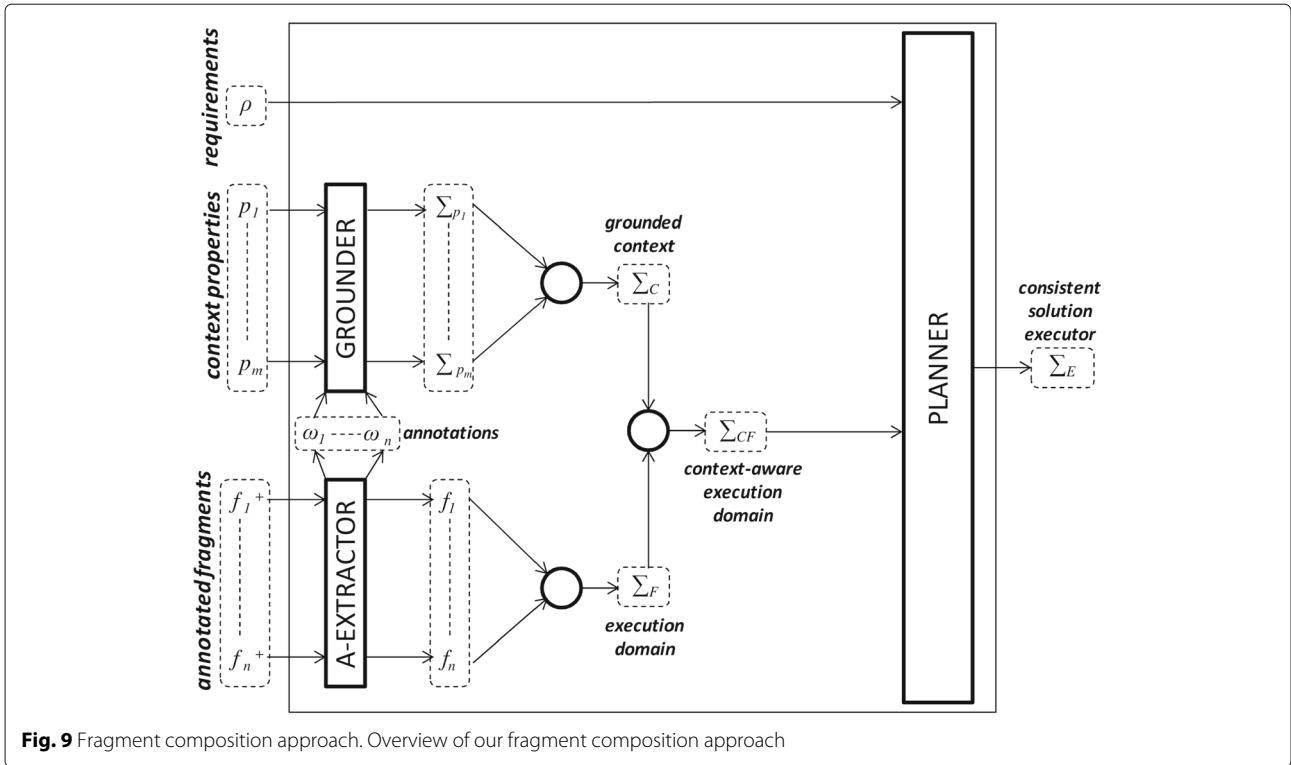
It is worth to notice that in this way it becomes much easier to introduce new fragments to a scenario at run time: it is enough that new fragments are properly annotated, while it is not necessary to change the context model

nor composition requirements<sup>5</sup>. As we will show in the next Section, once context properties and composition requirements are specified and component fragments are properly annotated, the whole set of specifications can be converted into a planning problem which is then resolved using planning algorithms. The obtained plan encodes a process that, if executed from the current context state, brings the system to one of the goal context configurations. The plan is further translated into an executable APFL process.

### 3.3 Fragment composition via planning

The overview of our fragment composition approach is given in Fig. 9. The composition engine accepts as input a context  $C$  represented by context properties  $p_1, p_2, \dots, p_m$ , a set of fragments  $F^+ = \{f_1^+, f_2^+, \dots, f_n^+\}$  annotated over  $C$  (together  $C$  and  $F^+$  form a context-





**Fig. 9** Fragment composition approach. Overview of our fragment composition approach

aware system  $\Psi = \langle F^+, C \rangle$  and composition requirements  $\rho$  expressed as a context formula over  $C$ . The output is an executor  $\Sigma_E$  that is a consistent solution executor for  $\Psi$  and  $\rho$ .

The focus of this article is to extensively present the service composition approach under the assumption that the set of eligible fragments for a specific composition problem has been already identified at the time of the composition. The selection of relevant fragments but also their reuse in similar composition problems is a very relevant aspect to make the composition approach more scalable.

Our implementation includes this optimization component and the complete formalization has been already presented in [8]. In it, we have defined a way to reduce the complexity of a planning problem (for a specific composition) by minimizing the search space according to the specific execution context, and reusing solutions (i.e., selected fragments) by learning from past executions. To select the relevant fragments in a specific context, QoS-aware service selection approaches as [15] can be also used. Another important aspect to highlight is that the composition approach proposed can be used not only

```

1  function plan(I, G)
2    OldSA :=  $\perp_{SA}$ 
3    SA :=  $\emptyset$ 
4    while (OldSA  $\neq$  SA  $\wedge$  I  $\notin$  (G  $\cup$  StatesOf(SA)))
5      Pr := StrongPreImage(G  $\cup$  StatesOf(SA))
6      NewSA := PruneStates(Pr, G  $\cup$  StatesOf(SA))
7      OldSA := SA
8      SA := SA  $\cup$  NewSA
9    done
10   if (I  $\in$  (G  $\cup$  StatesOf(SA)))
11     return SA
12   else
13     return  $\perp_{SA}$ 
14   fi
    
```

**Fig. 10** (Backward) Planning Algorithm. Algorithm for strong planning in asynchronous domain

to reach a specific goal of a certain *abstract* activity but also to repair or recompose broken service compositions [16, 17], as a reaction to context changes (i.e., precondition violations or service unavailability).

The very general idea of the approach consists in building a planning domain  $\Sigma_{CF}$ , that together with goal  $\rho$  form a planning problem. While, the resolution of a specific planning problem is carried out by an existing algorithm planning [18], the novel part of this article is the definition and the formalization of a composition problem that takes into account the system context at a specific execution time. To make it possible, the *execution domain*  $\Sigma_F$  is built as asynchronous product of fragments  $F^+$ . We assume that fragments within a single context-aware system have uncorrelated actions, i.e., such fragments have mutually disjoint sets of actions. In order to encode all possible parallel executions of fragments for some context-aware system we introduce the notion of *execution domain*, which is a parallel (asynchronous) product of fragments:

**Definition 6** (Execution Domain) *Let  $f_1 = \langle S_1, s_1^0, I_1, O_1, R_1 \rangle$  and  $f_2 = \langle S_2, s_2^0, I_2, O_2, R_2 \rangle$  be two observable state transition systems such that  $(I_1 \cup O_1) \cap (I_2 \cup O_2) = \emptyset$ . An execution domain  $\Sigma_F$  for fragments  $F = \{f_1, f_2\}$  is an asynchronous product of two fragments:*

$$\Sigma_F = \langle S_1 \times S_2, (s_1^0, s_2^0), I_1 \cup I_2, O_1 \cup O_2, R_F \rangle$$

where:

$$\begin{aligned} ((s_1, s_2), a, (s'_1, s'_2)) &\in R_F, \text{ if } (s_1, a, s'_1) \in R_1 \\ ((s_1, s_2), a, (s_1, s'_2)) &\in R_F, \text{ if } (s_2, a, s'_2) \in R_2 \\ R_F &= \{R_1 \parallel R_2\} \end{aligned}$$

Using fragment annotations  $(\omega_1, \omega_2, \dots, \omega_n)$  extracted by *A-EXTRACTOR*, context properties are grounded on fragment actions by *GROUNDER* so that the grounded context properties  $\Sigma_{p_1}, \Sigma_{p_2}, \dots, \Sigma_{p_m}$  are produced.

The grounding procedure, consists in replacing event-labeled transitions in context properties with action-labeled transition, which allows us to reflect the impact of fragment actions on the property state. Additionally, we use transition guards to reflect the executability of actions. As a result, the *grounded context property* features the same set of states as the original context property, but has different transition relation:

**Definition 7** (Grounded Context Property) *Let  $\Psi = \langle F^+, C \rangle$  be a context-aware system and let  $R_C$  be a space of context formulas of context  $C$ . A grounded context property for context property  $p = \langle L, l^0, E, T \rangle$  is a tuple  $\Sigma_p = \langle L, l^0, A_F, T^g \rangle$ , where:*

- $L$  is a set of states and  $l^0 \in L$  is the initial state;
- $A_F$  is a set of all fragment actions of fragments  $F^+$ ;
- $T^g \subseteq L \times R_C \times A_F \times L$  is a guarded transition relation.

The grounding procedure consists in defining a grounded context property  $\Sigma_p$  on top of a context property  $p$ . While the sets of states in  $p$  and  $\Sigma_p$  are the same, in  $\Sigma_p$  the event-based transitions of  $p$  are replaced with action-based transitions as indicated by annotations. For each transition of  $p$  labelled with event  $e$  and for each action  $a$  whose effect contains  $e$ , we define a transition in  $\Sigma_p$  with the same initial and final state and labelled with  $a$ . For each goal-labeled action  $a_{abs}$ , if an action goal (which is a conjunctive clause) requires that this property has to be in a particular state  $l$  (i.e., a proposition corresponding to  $l$  appears in the conjunctive clause expressing the goal of  $a_{abs}$ ), for every state in  $\Sigma_p$  we add a transition that starts in this state, terminates in  $l$  and is labelled with  $a_{abs}$ . Finally, for each action  $a_{less}$  that has no impact on the property and for each state in  $\Sigma_p$  we define a transition that starts and finishes in this state and is labelled with  $a_{less}$ . As such, we reflect the impact of all actions with respect to context property  $p$ . In order to take into account action preconditions, for each transition we introduce the guard, which is a precondition formula of its labelling action. A transition guard must be interpreted as a condition on the state of the whole context for which the transition is enabled. Formally:

**Definition 8** (Grounding) *Let  $\Psi = \langle F^+, C \rangle$  be a context-aware system. A grounding of a context property  $p = \langle L, l^0, E, T \rangle \in C$  is a grounded context property  $\Sigma_p = \langle L, l^0, A_F, T^g \rangle$  such that for every action  $a \in A_F$ :*

1. if  $\exists e \in E(a) : e \in E$  then for every transition  $(l, e, l') \in T$  there exists transition  $(l, P(a), a, l') \in T^g$ ;
2. if state  $l_g \in L$  appears in conjunctive clause  $G(a)$  then for every state  $l \in T$  there exists transition  $(l, P(a), a, l_g) \in T^g$ ;
3. if  $E(a) = \emptyset \wedge G(a) = \emptyset$  or  $(E(a) \neq \emptyset) \wedge (E(a) \cap E = \emptyset)$  or  $(G(a) \neq \emptyset) \wedge (\exists l_g \in L : l_g \in G(a))$  then for every state  $l \in L$  there exists a transition  $(l, P(a), a, l) \in T^g$ ;
4. no other states and transitions belong to  $\Sigma_p$ .

Since action effect contains no more than one event per context property, and since a goal conjunctive clause cannot contain more than one state per context property, the grounded context property is a deterministic LTS (only one transition with the same label is possible from each state).

In order to reflect the impact and executability of fragment actions with respect to the whole context we introduce the notion of *grounded context*, which is a synchronous product of all constituent grounded context

properties. We remark that the guards in the synchronous product can be removed. Indeed, for any guarded transition we can unambiguously figure out if the initial state of a transition satisfies the guard. Consequently, if the initial state satisfies the guard it is always 'unlocked' and we can replace it with the unguarded transition with the same properties, and if the initial state does not satisfy the guard it is always 'locked' and can be removed from the transition relation. Formally:

**Definition 9** (Grounded Context) *Let  $\Psi = \langle F^+, C \rangle$  be a context-aware system with context  $C = \{p_1, p_2, \dots, p_n\}$  and let  $\Sigma_{p_1}, \Sigma_{p_2}, \dots, \Sigma_{p_n}$  be the respective grounded context properties such that  $\Sigma_{p_i} = \langle L_i, l^0_i, A_F, T_i \rangle$  for all  $i \in [1, n]$ . Grounded context for  $\Psi$  is an LTS  $\Sigma_C = \langle L_C, l^0_C, A_F, T_C \rangle$  which is defined as follows:*

$$\Sigma_C = \langle L_1 \times \dots \times L_n, \{l^0_1, \dots, l^0_n\}, A_F, T_C \rangle$$

where:

$$((l_1, \dots, l_n), a, (l'_1, \dots, l'_n)) \in T_C, \quad (1)$$

$$\text{if } (l_i, P(a), a, l'_i) \in T_i \text{ for all } i \in [1, n] \quad (2)$$

$$\text{and } (l_1, \dots, l_n) \models P(a) \quad (3)$$

In turn, the context-aware execution domain can be constructed as a synchronous product of grounded context and execution domain.

Composition requirements  $\rho$  are compliant with the context-aware execution domain. By directly applying the planning algorithm to planning domain  $\Sigma_{CF}$  and planning goal  $\rho$ , we obtain the plan that is a consistent solution executor for a composition problem of  $\Psi$  and  $\rho$ . Moreover, if such plan is not found, then a consistent solution executor for a given composition problem does not exist.

### 3.3.1 Algorithm

For our convenience in this section we will omit the indices and denote the domain as follows:  $D = \langle S, s^0, I, O, R \rangle$ . The initial state of  $D$  becomes the initial state of the planning problem  $I = s^0$ , and the goal states are all states of the domain that satisfy  $\rho$ , that is  $G = \{s \in S : s \models \rho\}$ . As such, we obtain a conventional planning problem  $\{D, I, G\}$ .

Once a planning problem is obtained, a consistent solution executor is derived by the algorithm for strong planning in asynchronous domain presented in [18]. In the following, we briefly recap the description of this algorithm and definitions of theorems proving its termination, correctness and completeness.

The routine for searching a consistent solution executor is presented in Fig. 10. In this routine, we assume that the domain  $D$  is globally available, while we explicitly pass its initial states  $I$  and goal states  $G$ . The algorithm is a greatest fixed point iteration that incrementally constructs a

state-action table  $SA$ , which indicates which action has to be executed in certain state of  $D$  in order to reach a goal state. As such,  $SA$  encodes all transitions of the domain that can potentially be presented in the consistent solution executor.  $SA$  is initially empty and grows at each iteration by adding state-actions which unconditionally lead to the states that are already covered by  $SA$  or goal states (i.e., states  $\text{STATESOF}(SA) \cup G$ ). The termination of the algorithm is caused by either the situation when 1) no new states are included in the next iteration or 2) the current state-action table already contains all initial states  $I$ , which actually means that the solution for the initial states is already found.

The algorithm is defined such that it explicitly deals with the constraints imposed by a consistent solution executor. This logic is essentially realized by the key primitives **STRONGPREIMAGE** and **PRUNESTATES**.

**STRONGPREIMAGE** is the basis of the backward search. For a subset  $S$  of states of  $\Sigma_{CF}$ , **STRONGPREIMAGE** returns a set of state-action pairs  $\{(s, a)\}$  that encode all transitions of  $\Sigma_{CF}$  that immediately lead to  $S$ . It takes into account that uncontrollable actions can be neither controlled nor predicted. So the function guarantees that ones a state-action  $(s, a)$  is included in the table, states of  $S$  can always be reached from  $s$  despite non-determinism.

The primitive is defined as follows:

$$\text{STRONGPREIMAGE}(S) =$$

$$\{(s, a) : (a \in I) \wedge (\exists (s, a, s') \in R : s' \in S) \wedge \quad (4)$$

$$(\nexists (s, a', s'') \in R) : a \in O\} \cup \quad (5)$$

$$\{(s, a) : (a \in O) \wedge (\exists (s, a, s') \in R : s' \in S) \wedge \quad (6)$$

$$\forall (s, a', s'') \in R : (a' \in O) \rightarrow (s'' \in S)\} \quad (7)$$

In order to properly reflect the requirements imposed by the definition of consistent solution executor, controllable and uncontrollable actions are treated differently. For example, when we include controllable state-action, not only do we check that it leads to the states that are already in the state-action table but also make sure that uncontrollable actions are not available from the same state. Similarly, the way we treat uncontrollable actions guarantees that none of the uncontrollable actions originating from the same state are disregarded. Consequently, the strong pre-imaging function significantly contributes to the satisfaction of condition 1 (executor is runnable) of consistent solution executor. We remark that this planning algorithm is significantly different from the conventional strong planning algorithms (e.g., [19]) that treat all the actions of the planning domain uniformly.

**PRUNESTATES** function is responsible for removing from the current pre-image all the state for which the

solution is already available (i.e., those that are already included in the state-action table). It is defined as follows:

$$\text{PRUNESTATES}(\gamma, S) = \{(s, a) \in \gamma : s \notin S\}.$$

We remark that the purpose of the pruning goes beyond avoiding the duplication of the same state-actions in the resulting table. The pruning ensures that for each state no more than one controllable state-action is included which closely relates to condition 1 of consistent solution executor. It also guarantees that the state-action table does not contain loops (conditions 3). Another property of the pruning that has nothing to do with the definition of consistent solution executor is that only the shortest solution from any state appears in the state-action table.

The resulting *state-action table* (a collection of state-action pairs) shows how the resulting executable process should behave in different states. Uncontrollable state-actions indicate which uncontrollable actions have to be expected in the respective state. Similarly, controllable state-actions indicate which controllable action has to be executed from the respective state. The consistent solution executor  $\Sigma_{SA}$  can be directly derived from the state-action table using forward analysis.

For the given algorithms the following theorems can be proved (the respective proofs can be found in [18]).

**Theorem 1 (Termination)** *Let  $D = \langle S, s^0, I, O, R \rangle$  be a context-aware execution domain, let  $I = s^0$  be its initial*

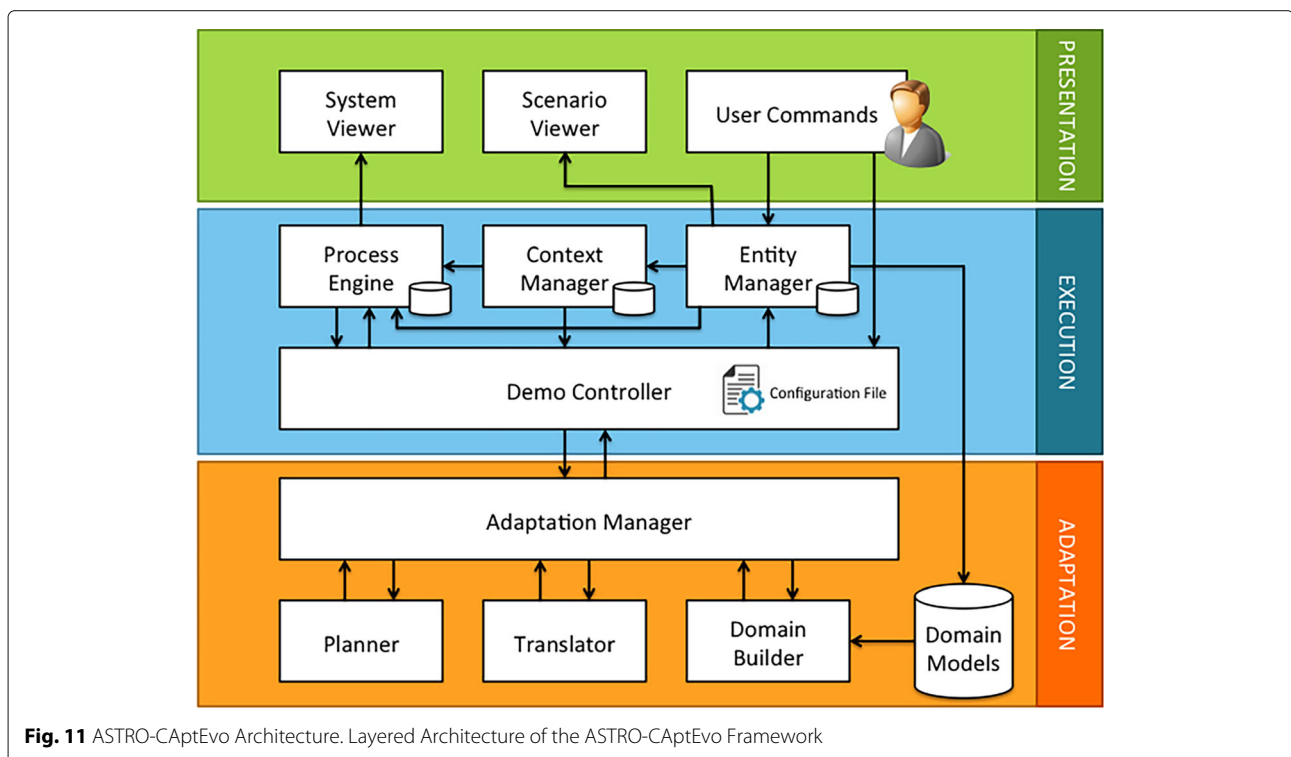
*state and let  $G \subseteq S$  be a set of goal states. The execution of  $\text{PLAN}(I, G)$  on  $D$  terminates.*

**Theorem 2 (Correctness and Completeness)** *If  $\text{PLAN}(I, G)$  returns state-action table  $SA$ , then  $\Sigma_{SA}$  is a consistent solution executor for the respective composition problem. If  $\text{PLAN}(I, G)$  returns FAIL, then no consistent solution executor to the respective composition problem exists.*

### 3.4 Framework implementation

The framework introduced in the previous sections has been implemented and is part of an extended version<sup>6</sup> of the ASTRO-CaptEvo framework [20, 21]. Its architecture is depicted in Fig. 11 and is composed by three layers, namely the Presentation, Execution and Adaptation layers.

The Adaptation layer is where the composition approach proposed in this paper has been implemented. The operation in this layer is regulated by the *Adaptation Manager*. It is notified about the need to refine an abstract activity with its respective goal, together with the information on the current context. All these information are passed to the *Domain Builder*. The *Domain Builder* builds an initial version of the composition problem, which is made of a context model, a set of available annotated fragments, the current context configuration (current states of context properties), and a set of goal context configurations. The *Domain*



**Fig. 11** ASTRO-CaptEvo Architecture. Layered Architecture of the ASTRO-CaptEvo Framework

*Builder* extracts all necessary specification from the repository of *Domain Models*. Taking into account the current context and the composition goal, the *Domain Builder* simplifies the context model by pruning all unreachable configurations and removing all fragments that are useless for the goal specified (see [8] for details on this pruning step). Thanks to this optimization phase, the size of the planning domain is reduced and the computation time for the planning phase is significantly reduced.

The *Translator* translates a composition problem into a planning problem such that it can be resolved by the *Planner*. It is also responsible for interpreting the results of the *Planner*. The back translation transforms a plan obtained into an executable process. Finally, the executable process is sent to the *Demo Controller* that injects it into the current process instance. All these steps are implemented in our framework and are graphically represented in the sequence diagram depicted in Fig. 12.

The Execution layer is in charge of 1) simulating the application domain, 2) executing process instances, 3) detecting when to call the Adaptation Layer to realize fragments composition, and 4) refining a process instance according to the solution received from the *Adaptation layer*.

The *Entity Manager* manages all active entities within the scenario (e.g., ships, cars, tracks, storage managers, etc.) and it simulates their behaviour. When the *Entity Manager* creates a new entity (either within the initialization phase, or in response to a user command), it deploys the entity process to the *Process Engine*, it adds the corresponding context properties to the context model in the *Context Manager* and it puts all the entity-related specifications (such as fragment models and the context property diagrams provided by the entity) to the *Domain*

*Models Repository*. When the entity “exits” the scenario, contrary actions are performed. In between, the *Entity Manager* simulates the entity behaviour, which is synchronized with the execution of the entity process, and updates the *Context Manager* and the *Scenario Viewer* with the current status. Finally, the *Entity Manager* processes all the user commands bringing changes to the domain (e.g., creation of new ships and orders, damage of a car, unavailability of fragments, etc..).

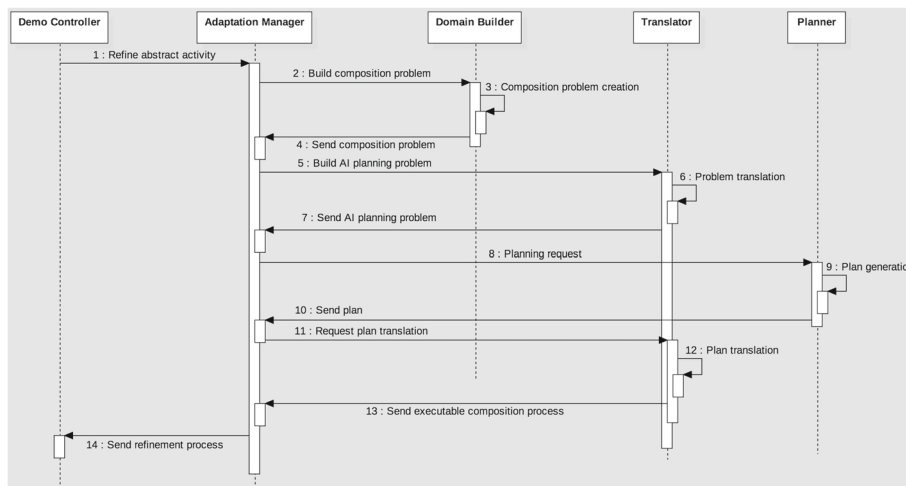
The *Context Manager* stores the system context (i.e., a set of context properties of all active entities) and it constantly synchronizes its current configuration with the application domain by monitoring the simulation going on in the *Entity Manager*.

The *Process Engine* executes the entity processes and suspends them every time that an abstract activity is reached and need to be refined. When it happens, the need is reported to the *Demo Controller*.

The *Demo Controller* aggregates all the data that is needed by the *Adaptation layer* to manage an abstract activity refinement (i.e., composition). This includes the current context provided by the *Context Manager* and the description of the need (process instance affected, its execution status, type of need) provided by the *Process Engine*. After the data is sent to the *Adaptation layer* and the solution is returned back, the *Demo Controller* adapts the process instance.

Finally, the adapted process instance is redeployed and restarted by the *Process Engine*.

The Presentation layer provides a detailed live view of the simulation taking place in the *Execution layer*. It also gives some control over the simulation and lets the user affect the application domain to model different situations.



**Fig. 12** ASTRO-CaptEvo: Fragments Composition approach. Sequence diagram that shows the dynamic part of the fragments composition approach

In particular, The *Scenario Viewer* provides a graphical representation of the application domain through a map containing all the facilities (landing gates, storage areas, roads etc.) and showing all the entities of the domain (cars, ships, managers, etc.) in action (see Fig. 13).

For each active entity, the *System Viewer* gives access to 1) the list of provided fragments and 2) the list of process instances. In turn, each process instance can be examined in a *Process Viewer* window (Fig. 14). Here the user can find:

- the process context information including all context properties and their current values;
- the process model with the execution progress;
- the execution history including all the applied compositions;

If the user is interested in how a certain abstract activity has been refined, the *Composition Inspector* (Fig. 15 provides full report from composition goals and fragments selection, to the details of the planning phase (planning domain, a resulting process, etc.).

The *User Commands* are used to control the simulation running in the *Execution layer* and bring changes to the running scenario. The user can affect the scenario by triggering exogenous events (e.g., unavailability of entities and fragments) and creating new entities. To

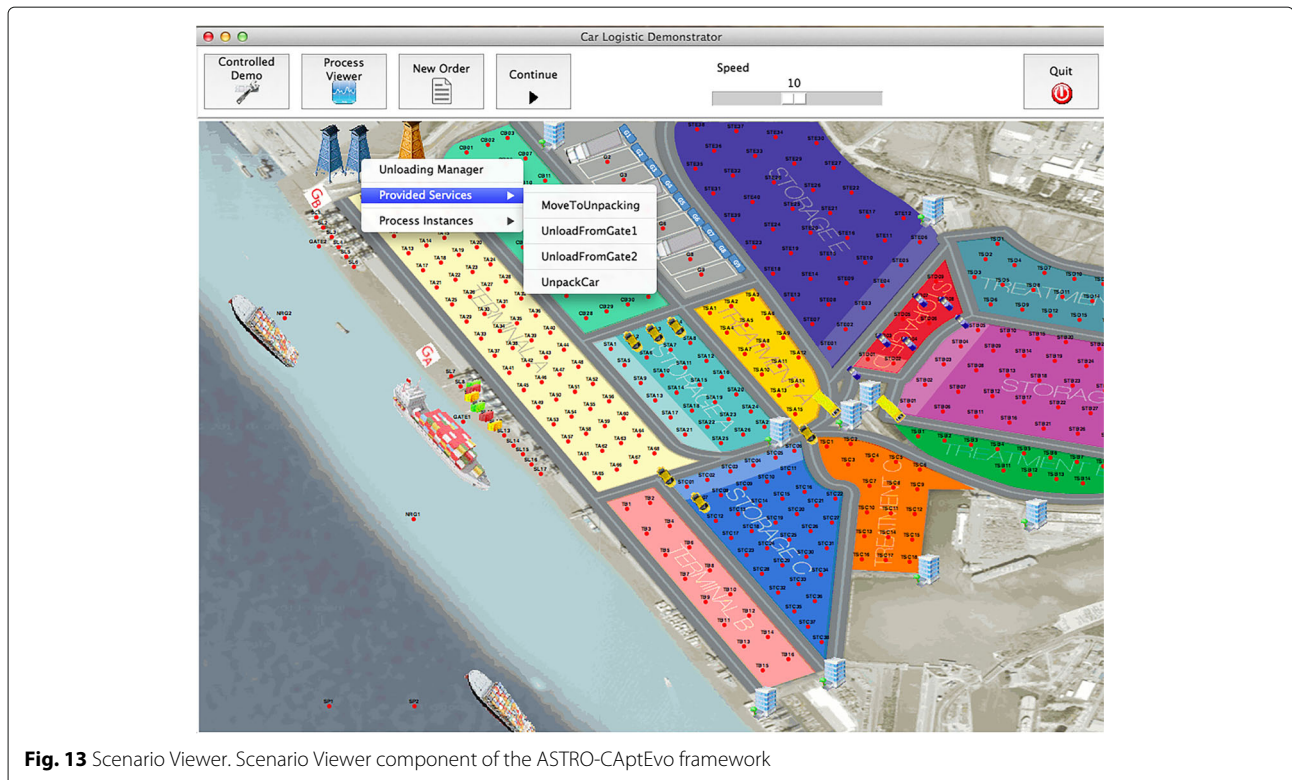
replicate the usage of the framework, in other application domains it is enough to: (1) define all the scenario models (entities, processes, fragments and context properties) that will be saved in the *Domain Models* repository; (2) define a *configuration file* that includes the specification of which types of entity must be instantiated and executed at simulation time. While all the other components of Fig. 11 can be executed without domain-specific extensions, the *Scenario Viewer* must be implemented from scratch, since it should provide a graphical representation of the application domain (e.g., as the harbour map in the Car Logistics).

#### 4 Experiments and results

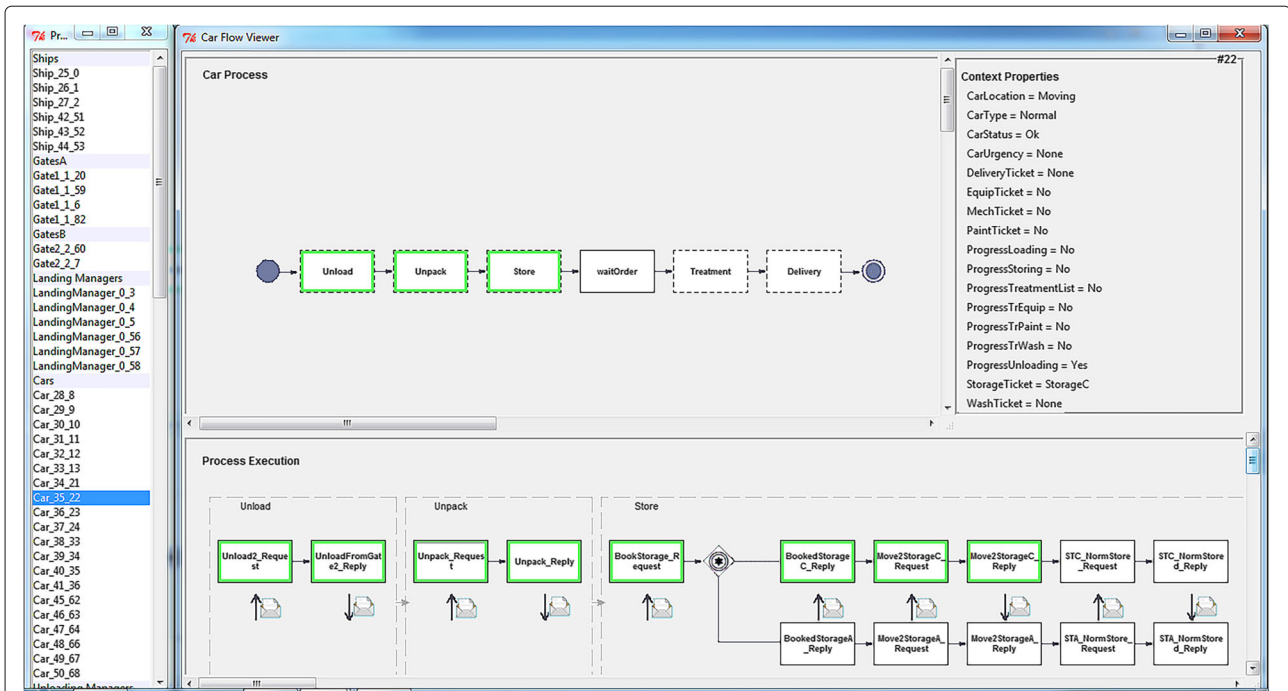
To demonstrate our approach in action and evaluate its performance and scalability, we have used the implementation presented in Section 3.4 to model and run the Car Logistics scenario introduced in Section 2. In the following sections we present the outcome of the set of experiments done.

##### 4.1 Experimental evaluation

We ran the ASTRO-CAptEvo framework in continuous mode for around an hour and collected information on 1060 compositions performed within this time. The run was performed on a Windows laptop with dual-core 2.8GHz CPU with 8Gb of RAM (we remark, however, that



**Fig. 13** Scenario Viewer. Scenario Viewer component of the ASTRO-CAptEvo framework



**Fig. 14** Process Viewer. Process Viewer component of the ASTRO-CaptEvo framework

the planner implementation is single-threaded). For each composition, we measured a number of indicators that characterized the complexity of the problem and the timing. Then we tried to organize them into charts to make conclusion about the applicability of the approach.

While conducting the measurements, we took into account the general conclusion about the performance of the planning algorithm given in [18]. In particular the authors stated that “the performance of synthesis appears to degrade sub-exponentially with the size of the components; and in vast majority of cases, it degrades polynomially with the number of components”.

However, when working with bigger domains the performance may degrade to exponential. The most reasonable explanation for that is based on the implementation details of the BDD (binary decision diagram) library used: big domains are much more memory demanding and for them the garbage collection and data re-arrangement mechanisms may take considerable time to keep the memory consumption within certain limits.

In the chart in Fig. 16 we show the dependency between the number of fragments passed to the planner and the time it takes to produce a plan (in logarithmic scale). It can be seen that it shows exponential scalability. In general, this result corresponds to that of [18]. The performance degradation to exponential even for small numbers of fragments (that was not present in [18]) can be explained by the fact that, in addition to fragments, the planning domain in context-aware composition also contains

context-related LTSs. As a result, even for small number of fragments, the domain becomes relatively large and results in exponential scalability.

Alternatively, we propose our own indicator of domain complexity that is the total number of transitions in fragments and context properties making up the domain:

$$Complexity = NumContextTrans + NumFragTrans.$$

We find this indicator more precise compared to the number of fragments. It also allows us to see a more fine-grained distribution of all composition problems with respect to complexity. The performance scalability with respect to composition complexity is represented by the chart in Fig. 17. It can be observed that it generally corresponds to the chart in Fig. 16 and features exponential growth. However, it makes sense to consider this chart along with the complexity distribution of all composition problems analyzed in Fig. 17. It can be observed that most composition cases reside in the region with low or moderate complexity, while the cases with high complexity are quite few. We remark that such distribution also affects the precision of the scalability chart in the region of high complexity (less experiments are carried out there).

Consequently, from the charts in Fig. 17 we can derive the following table showing the percentage of composition cases that are resolved in no more than *n* seconds:

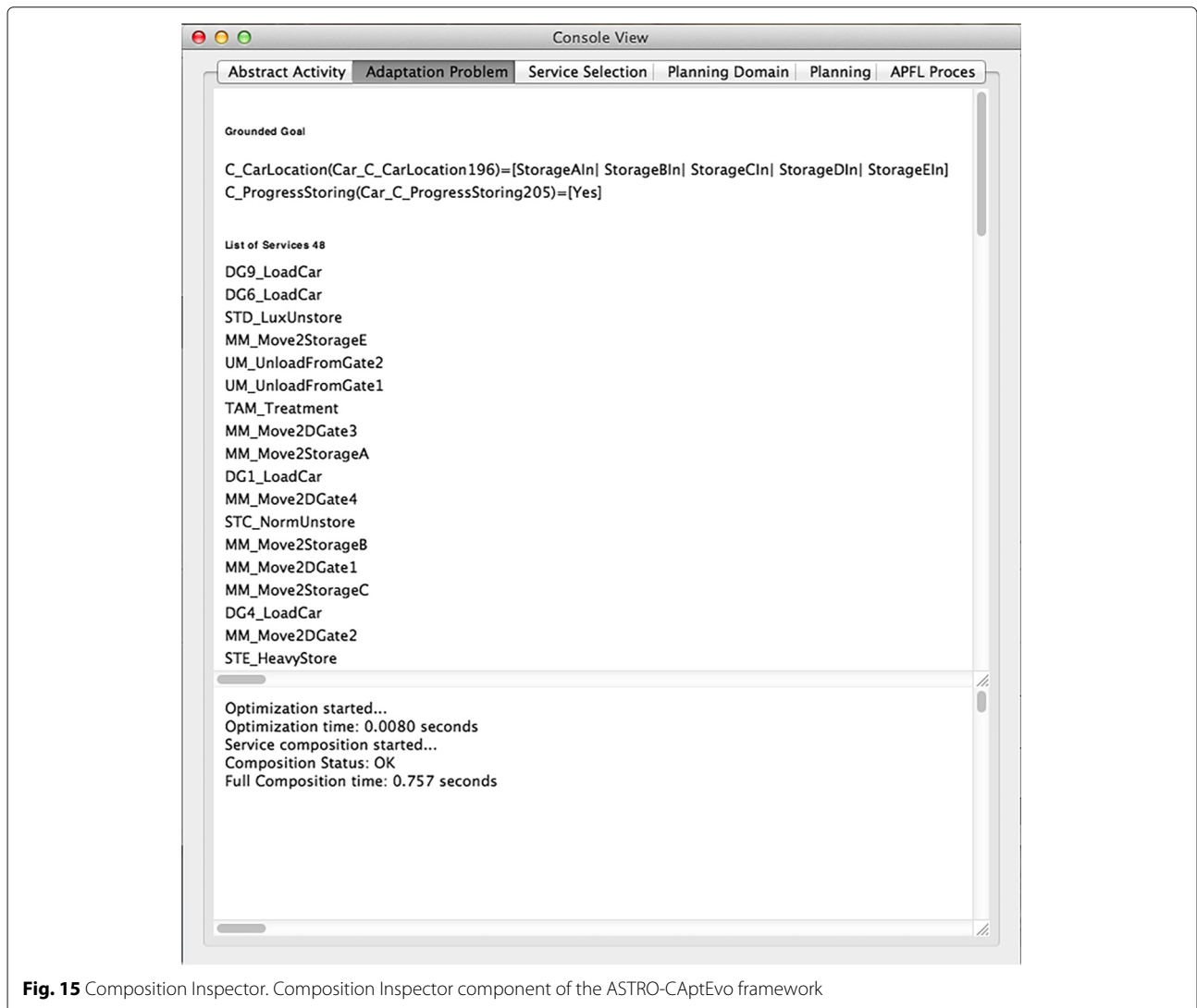


n, sec	compositions resolved within n sec, %
0.1	19.07
1	91.12
3	96.51
10	99.62
30	100.00

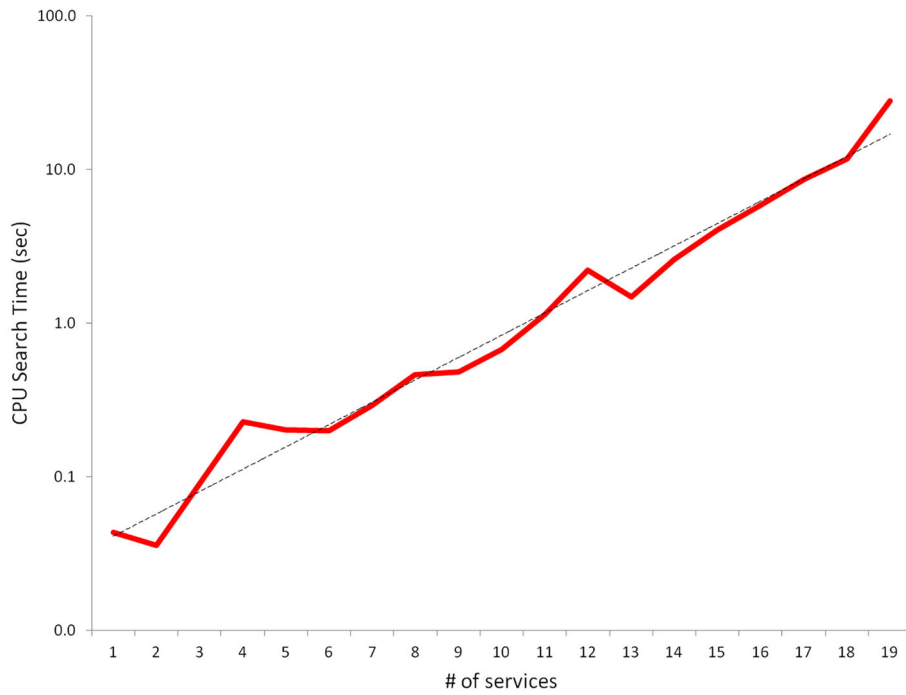
From the table it can be observed that the vast majority of adaptation-related compositions actually take less than 10 seconds. This is the first evidence of practical applicability of our composition technique: although the performance of context-aware composition degrades exponentially with growing complexity of a composition problem, it is still enough to be used for the actual problem of process adaptation. This becomes especially true when we notice that in many application domain there are no severe restrictions on the performance of composition

related tasks. For example, in the CLS scenario, the typical life cycle of a car may have duration up to several weeks and usually it is affordable for a user-centric system to take up to half a minute to produce a solution.

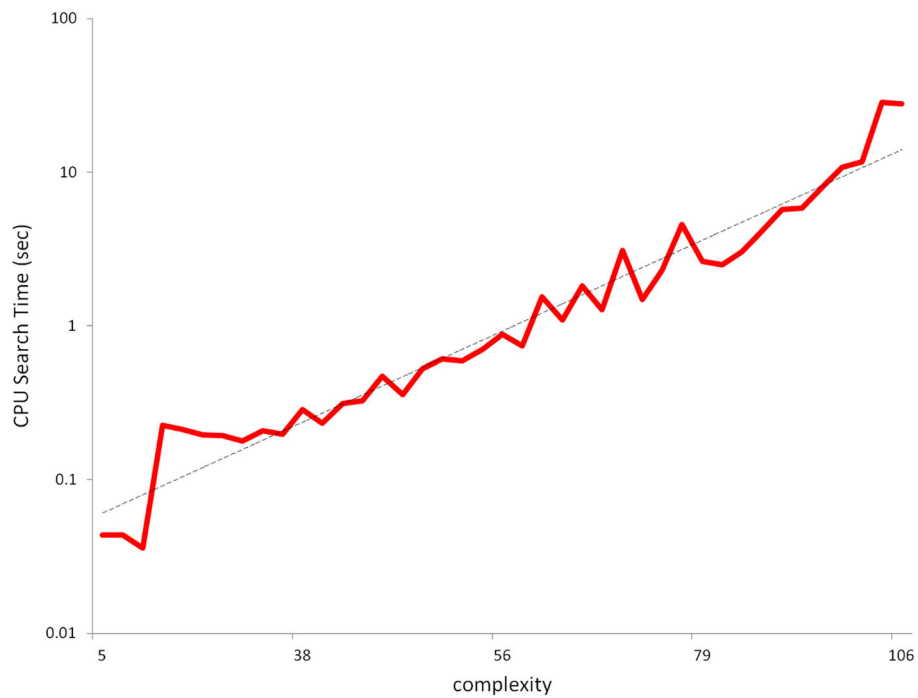
The last important observation is that for each particular composition problem we build a planning domain that includes only the information that is relevant for this problem, namely: 1) the subset of context properties that are relevant for entities under consideration, which is normally a small portion of the overall context of the scenario and 2) the subset of all fragments that may be useful within the current composition problem, which is, again, only a small portion of all fragments currently available in the system. The idea of fragments and context pre-selection is quite natural: if the system resolves a problem for a particular car it needs only the part of context that are relevant for this car (and not for dozens of other cars and ships in the system), and fragments that



**Fig. 15** Composition Inspector. Composition Inspector component of the ASTRO-CAPT Evo framework



**Fig. 16** Dependency between performance and number of fragments composed. Dependency between the number of fragments passed to the planner and the time it takes to produce a plan



**Fig. 17** Complexity distribution and performance scalability. The performance scalability with respect to composition complexity

are relevant for car only (e.g., it does not make sense to consider a fragment for ship landing). We expect this pre-selection to allow us to preserve the same average size of the planning problem even for much larger (with respect to the number of entities) domains. Indeed, if within the scenario we operate thousands of cars instead of dozens, the complexity of an average composition problem for a particular car would not grow and remain the same: the proper selection will always come up with more or less the same amount of relevant fragments and context. The increase in the amount of cars in the harbour does not functionally affect the way a car procedure should be planned and executed. As such, we expect our approach to be easily scalable in this regard.

## 5 Related work

In last years, different approaches have been proposed for the modeling of services in a suitable way for making efficient dynamic service composition. From the scenario and the challenges discussed in this paper, a specific need has emerged: being able to define services in such a way that they can dynamically be specialized to the context, when this is discovered or when it changes. Where the context is characterized by the state of the execution environment and by the available services at a specific time and location.

An approach targeting this problem is presented by Hull et al. [22] in their work about *Business Artifacts*. It consists in the definition of a formal/theoretical framework for defining conceptual entities, the artifacts, related to the execution of services whose operations influence the entities evolution, as they move through the business's operations. However, this approach mostly focuses on service modeling aspects and does not deal with dynamic service composition. Yu et al. propose MoDAR [23], an approach on how to design dynamically adaptive service-based systems. Essentially they propose a method to simplify business processes by using rules to abstract their complex structure and to capture their variable behavior. However, in dynamic context, revising rules to manage frequent and unpredictable changes might turn out to be very expensive and complex. In [24], the authors tackle the problem of unpredictable execution of service-based applications. In particular, they focused on how to evolve a running service composition and propose a way for modeling artifacts corresponding to composite services that can change at runtime. However, software engineer intervention is needed to manipulate the runtime model of services. Moreover, the adaptation and application logics are mixed making the model not very flexible. In [25] the authors present DAMASCo, a framework managing the discovery, composition, adaptation and monitoring of services in a context-aware fashion by taking into account semantic information rather than only the syntactic one.

Since they address the problem of making the reuse of software entities more agile and easy to model, they focus especially on the adaptation of pre-existing software entities that are used during the developing of service-based applications. Also the approach presented in [26] focuses on the need of explicitly manage the context in the composition of web services, to address the problem of semantic heterogeneities between services. The authors present a context-based mediation approach that allows services both to share a common meaning of exchanged data and to automatically resolve conflicts related to the semantic of the data, by using context-based annotations which offer an optimized handling of the data flow. It would be interesting to use the approaches [25, 26] in the management of the composition of fragments coming from the different entities and the definition of the data flow between them. In [27] the concepts of goals and plans are introduced in the business processes modeling with the purpose of extending the standard BPMN to make the BPM more flexible and responsive to change. However, even if plans are selected and executed at runtime, they are defined at design time together with the relations with the goals they can satisfy. Göser et al. [28] is a framework for, among other things, the management of the integration of services in the business processes implementation's process to speed the implementation and deployment phases. Services' integration is realized in a plug-and-play manner in which activities are selected from a repository and then dropped into a process. However, as regards the runtime adaptation of processes, in this approach only ad-hoc modifications are managed.

Hermosillo et al. [29] is a framework that combines complex event processing and dynamic business process adaptation, which allows to respond to the needs of the rapidly changing environment, and its adaptation language called SBPL, and extension to BPEL which adds flexibility to business processes. As in the previous framework, only ad-hoc adaptation processes are defined ad design-time together with the definition of specific adaptation points for the business process and the events that will trigger that adaptation.

In the context of Future Internet [30], some frameworks have been proposed to deploy and execute adaptable, QoS-aware service compositions. In [31], authors present an engine for the execution of service compositions based on a *unified model*. The unified model allows to execute service compositions that are specified by use of different languages with different underlying modeling paradigms, e.g. imperative and declarative service compositions by the same engine. Furthermore, the unified model and the presented engine enables the unification of the execution of service orchestrations and the enactment of service choreographies. CHOReOS

project [32] proposes a dynamic development process, and associated methods, tools and middleware, to sustain the composition of services in the Future Internet. It proposes a synthesis approach able to automatically generate, out of a BPMN2 choreography specification, the needed adaptation and coordination logic, and distribute it between the participants so to enforce the choreography. Finally, in [16], authors propose service composition repair as an alternative solution that goes beyond the limits of service replacement while avoiding recomposition.

## 6 Discussion

We conclude this section with a discussion (summarized in Table 1) in which we try to point out the advantages of the proposed approach respect to related works. As regards to the standard approaches of service composition, such as those of orchestration and choreography, they have some crucial limitations. A major problem of these approaches is that most of them are based on the assumption that during the specification of composition requirements, the application designer knows the services to be composed. Besides, some of them, such as [14, 33], remain focused on the syntax level without considering the semantic aspects of composition, which are, instead, necessary in context-based applications. Other approaches like [31, 34–38], have introduced the management of semantic knowledge in their models to drive the services' composition and interoperation but, despite this, they do not allow processes to be specialized at runtime, through dynamic service composition. Cubo and Pimentel [25], Mrissa et al. [26], Greenwood [27] allow for very efficient management of service compositions at runtime, while [16, 32] supports composition evolution through adaptation to possible changes in the discovered services. The adaptation strategies applied in these approaches in some cases are defined at design-time or are strongly

related to a prescribed coordination model (i.e., BPMN2 model in [32] does not open to runtime and context-aware refinements).

The approach proposed in this paper, instead, offers a lightweight-model, with respect to the existing languages for service composition. It is more flexible and able to define both orchestrations and choreographies thanks to its dynamic collaboration among entities. Moreover, the model explicitly handles the context by managing the dynamicity of services, which can enter or leave the system in any moment, with a flexible connection strategy between entities that exploit the *publish-subscribe paradigm*. Unlike specifications of traditional systems, where the behaviors are static descriptions of the expected run-time operation, our approach allows the application to define dynamic behaviors. This is realized thanks to the usage of *abstract activities* representing opening points in the definition of processes, which allow services being refined when the context is known or discovered. The *bottom-up approach* for the activities' refinement allows *fragments*, once they are selected for the composition, to climb the entities's relations to be embedded in the running process. Besides, the composition is defined at runtime, so that exactly the currently available services are considered for the composition. This feature, on the one hand enables a smooth exploitation of proximity services, and on the other hand makes the impact of run-time changes to services (modification of behavior, entrance or exit of services from the system) transparent to the system execution.

## 7 Conclusion

In this article, we presented a service composition framework that overcomes many limitations of the existing approaches. Our approach uses an AI planning algorithm as a reasoning mechanism, and can be used to solve composition problems of real-world complexity in dynamic

**Table 1** Service composition approaches comparison

Approaches	Customization and context-awareness	Openness	Flexibility
Our approach	<i>Runtime</i> service composition in a dynamic context	Transparent handling of new services available at <i>runtime</i>	Structural changes in services functionalities and services unavailability
[22–29]	Context-aware selection of services using semantic information, or context events to identify adaptation situations	Not addressed	Adaptation plans defined at <i>design-time</i> together with the relations with the goals they can satisfy or predefined adaptation points.
[14, 33]	Not addressed	Syntax level service selection and composition at <i>design-time</i>	Not addressed
[16, 31, 32, 34–38],	Semantic Knowledge to drive the service composition	<i>Design-time</i> services selection, binding and composition	Service Choreography evolution through adaptation to possible changes in the discovered services

and pervasive setting like Logistics [8] or Smart Urban Mobility [7] domains.

The proposed framework uses an innovative service model in which services are considered to be stateful, non deterministic and asynchronous. The composition requirements model is based on the idea of abstracting composition requirements from implementation details of services. This allows for deep automation of the composition process. Even though in the article we only consider reachability goals as control-flow requirements, we emphasize that our approach can be easily adapted to use any advanced control-flow language used in planning (e.g., [39]). We also remark that our approach is compatible with data-flow requirements technique introduced in [40]. All this makes the new composition engine applicable to a wide range of real service composition problems.

Thanks to the abstraction of composition requirements from implementation of components we organized the composition life-cycle in such a way that almost all the human time-demanding operations can be accomplished at design time so that the composition run-time is fully automated. Moreover, the availability of a tool with such essential property brings new possibilities to composition dependent systems.

We generally consider two main future steps in the development of the ideas presented. One of them naturally consists in adopting the advanced compatible techniques for specifying complex control- and data-flow requirements. Another one concerns conducting experiments on using our approach in user-centric systems.

## Endnotes

<sup>1</sup>In this article, we focus only on control-flow aspect of composition. Data-flow aspect can be handled using the technique presented in [40], which is compatible with the formal framework introduced.

<sup>2</sup>Although the proposed approach works with processes modeled using APFL, it can be also used extending other process-based languages like BPMN [41] or CMMN [42].

<sup>3</sup>We remark that in order to avoid state explosion and to keep a planning problem tractable, it is strictly required that the number of states of each context property is finite and reasonably small. One technique for dealing with context properties with large or infinite number of states in planning can be found in [43].

<sup>4</sup>In this article, as context formulas we use reachability goals over context states (i. e., a goal consists in achieving certain context states, known as goal states). At the same time, we emphasize that used AI planning

techniques support for more sophisticated constructs including procedural goals, reactive goals and goals with preferences (more details can be found in [39]).

<sup>5</sup>In our approach, the need to annotate newly created fragments to integrate them into the running system is the only human-dependent task that needs to be accomplished at run time. We do not see it as a significant limitation since 1) the annotation of new fragments is normally far less urgent task than resolving an ongoing failure and can hardly interrupt the normal operation of the system, and 2) the annotation effort can be distributed among multiple partners (each provider annotates its fragments).

<sup>6</sup><http://das.fbk.eu/astro-captevo>

## Abbreviations

APFL: Adaptable pervasive flows language (Section 4.1); BDD: Binary decision diagram (Section 5.2); CLS: Car logistics scenario (Section 3.1); IoS: Internet of services (Abstract); LTS: Labelled transition system (Section 3.1)

## Availability of data and materials

The implementation of the ASTRO-CaptEvo framework is available at <http://das.fbk.eu/astro-captevo> together with a video showing its live demonstration.

## Authors' contributions

All authors are equal contributors. All authors read and approved the final manuscript.

## Authors' information

Dr. Antonio Bucchiarone (PhD) is a senior researcher in the FBK-DAS research unit of FBK. He received his PhD in Computer Science and Engineering, from IMT of Lucca (Italy) in July 2008. His main research interests are: self-adaptive (collective) systems, applied formal methods, run-time service composition and adaptation, specification and verification of component-based systems, dynamic software architectures. He has been actively involved in various research projects in the context of self-adaptive systems.

Dr. Annapaola Marconi (PhD) is a senior researcher at FBK, where she directs the FBK-DAS research unit. She received her PhD in Computer Science in 2008, from the ICT International Doctoral School of the University of Trento. Her research interests include distributed systems, collective adaptive systems, and automated composition of service-based applications. She has been actively involved in various local and European research projects in the area of Smart Mobility.

Dr. Marco Pistore (PhD), is currently head of the FBK-SC unit. He received a PhD in Computer Science from the University of Pisa (Italy) in 1998. He has an h-index of 46 and more than 200 publications in international journals, conferences, and symposia. He has 12 years of experience in the management of research teams and projects: he has been responsible of research groups and project teams (up to more than 30 persons); he has been scientific coordinator and partner coordinator of regional, national and EU research and innovation projects; he has been responsible of technology transfer projects with National and International companies.

Dr. Heorhi Raik (PhD), is a junior researcher at FBK. He received his PhD in Computer Science in 2012, from the ICT International Doctoral School of the University of Trento, with a thesis titled: "Service Composition in Dynamic Environments: From Theory to Practice". His research interests include service composition, dynamic process adaptation, user-centric services, AI planning, collective adaptive systems.

## Competing interests

The authors declare that they have no competing interests.

## Publisher's Note

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

Received: 27 October 2016 Accepted: 28 April 2017

Published online: 05 June 2017

## References

- Autili M, Tivoli M, Goldman A (2016) Thematic series on service composition for the future internet. *J Internet Serv Appl* 7(1):3–134
- Issarny V, Georgantas N, Hachem S, Zarras A, Vassiliadis P, Autili M, Gerosa MA, Hamida AB (2011) Service-oriented middleware for the future internet: state of the art and research directions. *J Internet Serv Appl* 2(1):23–45
- Zhou J, Gilman E, Palola J, Riekkii J, Ylianttila M, Sun JZ (2011) Context-aware pervasive service composition and its implementation. *Pers Ubiquit Comput* 15(3):291–303
- Tari K, Amirat Y, Chibani A, Yachir A, Mellouk A (2010) Context-aware dynamic service composition in ubiquitous environment. In: *Proceedings of IEEE International Conference on Communications, ICC 2010*. IEEE, Cape Town. pp 1–6
- Truong HL, Dustdar S (2009) A survey on context-aware web service systems. *IJWIS* 5(1):5–31
- Bucchiarone A, de Sanctis M, Marconi A, Pistore M, Traverso P (2015) Design for adaptation of distributed service-based systems. In: *Service-Oriented Computing - 13th International Conference, ICSOC 2015, November 16–19, 2015, Proceedings*. Springer, Goa. pp 383–393
- Bucchiarone A, de Sanctis M, Marconi A, Pistore M, Traverso P (2016) Incremental composition for adaptive by-design service based systems. In: *IEEE International Conference on Web Services, ICWS 2016, June 27 - July 2, 2016*. IEEE, San Francisco. pp 236–243
- Bucchiarone A, Marconi A, Mezzina CA, Pistore M, Raik H (2013) On-the-fly adaptation of dynamic service-based systems: Incrementality, reduction and reuse. In: *Service-Oriented Computing - 11th International Conference, ICSOC 2013, December 2–5, 2013, Proceedings*. Springer, Berlin. pp 146–161
- Eberle H, Unger T, Leymann F (2009) Process fragments. In: *On the Move to Meaningful Internet Systems: OTM 2009, Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009, Vilamoura, Portugal, November 1–6, 2009, Proceedings, Part I*. Springer. pp 398–405
- Raik H (2012) *Service Composition in Dynamic Environments: From Theory to Practice*. PhD Dissertation. Available at <http://eprints-phd.biblio.unitn.it/864/>
- Böse F, Piotrowski J (2009) Autonomously controlled storage management in vehicle logistics applications of rfid and mobile computing systems. *Int J RT Technol Res Appl* 1(1):57–76
- Bucchiarone A, Lluch-Lafuente A, Marconi A, Pistore M (2009) A formalisation of adaptable pervasive flows. In: *Web Services and Formal Methods, 6th International Workshop, WS-FM 2009, September 4–5, 2009, Revised Selected Papers*. Springer, Bologna. pp 61–75
- Herrmann K, Rothermel K, Kortuem G, Dulay N (2008) Adaptable pervasive flows - an emerging technology for pervasive adaptation. In: *Workshop on Pervasive Adaptation (PerAda)*
- Committee OWT (2007) *Web services business process execution language, version 2.0*. Available at <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0>
- Haddad JE, Manouvrier M, Rukoz M (2010) Tqos: Transactional and qos-aware selection algorithm for automatic web service composition. *IEEE Trans Serv Comput* 3(1):73–85
- Yan Y, Poizat P, Zhao L (2010) Repair vs. recomposition for broken service compositions. In: *Service-Oriented Computing - 8th International Conference, ICSOC 2010, ebruary 7–10, 2010, Proceedings*. Springer, San Francisco. pp 152–166
- Bucchiarone A, Marconi A, Pistore M, Raik H (2012) Dynamic adaptation of fragment-based and context-aware business processes. In: *2012 IEEE 19th International Conference on Web Services, June 24–29, 2012, Honolulu*. pp 33–41
- Bertoli P, Pistore M, Traverso P (2010) Automated composition of web services via planning in asynchronous domains. *Artif Intell* 174:316–361
- Cimatti A, Pistore M, Roveri M, Traverso P (2003) Weak, strong, and strong cyclic planning via symbolic model checking. *Artif Intell* 1–2:35–84
- Bucchiarone A, Marconi A, Pistore M, Raik H (2012) Dynamic adaptation of fragment-based and context-aware business processes. In: *2012 IEEE 19th International Conference on Web Services, June 24–29, 2012, IEEE, Honolulu*. pp 33–41
- Raik H, Bucchiarone A, Khurshid N, Marconi A, Pistore M (2012) Astro-captevo: Dynamic context-aware adaptation for service-based systems. In: *Eighth IEEE World Congress on Services, SERVICES 2012, June 24–29, 2012, Honolulu*. pp 385–392
- Hull R, Damaggio E, De Masellis R, Fournier F, Gupta M, Heath III FT, Hobson S, Linehan MH, Maradugu S, Nigam A, Sukaviriya PN, Vaculin R (2011) Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In: *Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011, July 11–15, 2011*. ACM, New York. pp 51–62
- Yu J, Sheng QZ, Swee JKY (2010) Model-driven development of adaptive service-based systems with aspects and rules. In: *WISE. Lecture Notes in Computer Science, vol. 6488*. pp 548–563
- Hussein M, Han J, Yu Y, Colman A (2013) Enabling runtime evolution of context-aware adaptive services. *IEEE International Conference on Services Computing*
- Cubo J, Pimentel E (2011) Damasco: A framework for the automatic composition of component-based and service-oriented architectures. In: *Software Architecture - 5th European Conference, ECSA 2011, September 13–16, 2011, Proceedings*. Springer, Essen. pp 388–404
- Mrissa M, Ghedira C, Benslimane D, Maamar Z, Rosenberg F, Dustdar S (2007) A context-based mediation approach to compose semantic web services. *ACM Trans Internet Techn* 8(1)
- Greenwood DAP (2008) Goal-oriented autonomic business process modeling and execution: Engineering change management demonstration. In: *Business Process Management, 6th International Conference, BPM 2008, September 2–4, 2008, Proceedings*. Springer, Milan. pp 390–393
- Göser K, Jurisch M, Acker H, Kreher U, Lauer M, Rinderle S, Reichert M, Dadam P (2007) Next-generation process management with ADEPT2. In: *Proceedings of the BPM Demonstration Program at the Fifth International Conference on Business Process Management (BPM'07), 24–27 September 2007, Brisbane*. Springer
- Hermosillo G, Seinturier L, Duchien L (2010) Creating context-adaptive business processes. In: *Service-Oriented Computing - 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7–10, 2010, Proceedings*. pp 228–242
- Autili M, Goldman A, Tivoli M (2015) IEEE services visionary track on service composition for the future internet (SCFI 2015). In: *2015 IEEE World Congress on Services, SERVICES 2015, June 27 - July 2, 2015, IEEE, New York City*. pp 327–328
- Görlach K, Leymann F (2015) A flexible engine for the unified execution of service compositions. In: *2015 IEEE Symposium on Service-Oriented System Engineering, SOSE 2015, March 30 - April 3, 2015, San Francisco Bay, IEEE*. pp 133–142
- Autili M, Inverardi P, Tivoli M (2014) CHOREOS: large scale choreographies for the future internet. In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, February 3–6, 2014, Antwerp, IEEE*. pp 391–394
- Kavantzias N, Burdett GRD (2004) Wscdl v1.0. Available at <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>
- BPML.org (2002) *Business process modeling language (bpml)*. Available at <http://www.bpml.org>
- Arkin A, Askary S, Fordin S, Jekeli W, Kawaguchi K, Orchard D, et al (2002) *Web service choreography interface (wsci)*. Available at <http://www.w3.org/TR/wsci>
- McGuinness DL, van Harmelen F (2004) *Owl web ontology language overview* [online]. Available at <http://www.w3.org/TR/owl-features/>
- McIlraith SA, Son T, Zeng H (2001) Semantic web services. *IEEE Int Syst* 16(2):46–53
- WSMO WsMO working group. Available at <http://www.wsmo.org>
- Traverso P, Pistore M (2004) Automated composition of semantic web services into executable processes. In: *International Semantic Web Conference (ISWC)*. pp 380–394
- Kazhamiakin R, Marconi A, Pistore M, Heorki R (2013) Data-flow requirements for dynamic service composition. In: *Proceedings of the 20th International Conference on Web Services*. pp 243–250

41. Group OM (2011) Business process model and notation - version 2.0. Available at <http://www.omg.org/spec/BPMN/2.0/>
42. Group OM (2016) Case management model and notation (cmmn) - version 1.1. Available at <http://www.omg.org/spec/CMMN/1.1/>
43. Pistore M, Marconi A, Bertoli P, Traverso P (2005) Automated composition of web services by planning at the knowledge level. In: IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, July 30 - August 5, 2005, Edinburgh. pp 1252–1259

**Submit your manuscript to a SpringerOpen<sup>®</sup> journal and benefit from:**

- ▶ Convenient online submission
- ▶ Rigorous peer review
- ▶ Immediate publication on acceptance
- ▶ Open access: articles freely available online
- ▶ High visibility within the field
- ▶ Retaining the copyright to your article

---

Submit your next manuscript at ▶ [springeropen.com](http://springeropen.com)

---