Abstract—A critical aspect for pervasive computing is the possibility to discover and use process knowledge at run time depending on the specific context. This can be achieved by using an underlying service-based application and exploiting its features in terms of dynamic service discovery, selection, and composition. Pervasive process fragments represent a service-based tool that allows to model incomplete and contextual knowledge. We provide a solution to automatically compose such fragments into complete processes, according to a specific context and specific goals. We compute the solution by encoding process knowledge, domain knowledge and goals into an AI planning problem. We evaluate our approach on different scenarios stress testing the main characteristics of pervasive process fragments.

I. INTRODUCTION

Service-based applications are a promising solution for the dynamic and heterogeneous domain of pervasive computing. Service-based applications are compositions of software services, possibly offered by different third-party organizations. A pervasive computing environment is a physical environment populated by small networked processing devices that interact with users. In such a setting, it is critical to allow for processes to be discovered at runtime and to be used depending on the context (e.g., depending on location, time, situation, people). This can be achieved with an underlying service-based application, since the software services can be discovered, selected and composed dynamically, while the application is running.

An approach going in this direction is that of process fragments [1]. Process fragments represent a tool for modeling incomplete and local process knowledge. The knowledge is incomplete since the modeler is allowed to specify just one aspect of the entire process, and even to leave gaps in the process specification. Fragments can be modeled by different people, and therefore may reflect different perspectives on the same process. Further, the process knowledge is local, since the availability and usability of a fragment is determined by context. For example, the execution of a process fragment may be bound to a certain location.

Process fragment knowledge can be integrated dynamically at design time or at run time. This requires enriching processes with goals which specify what is pursued by the process execution. It also requires enriching fragments with information on how they contribute to the outcome of the process.

We apply the concept of process fragments to the concrete case of adaptable pervasive flows, introduced within the European project ALLOW (http://www.allow-project.eu). An adaptable pervasive flow is a dynamic workflow modeling the behavior and goals of a physical entity, such as an artifact or a person. Flows are context-aware: during their execution, they can sense the environment related to their entity. For this purpose, flows include special modeling elements explicitly representing context information. To distinguish the fragments of adaptable pervasive flows from process fragments, we will refer to the former as pervasive process fragments (or simply, fragments). The fact that they represent incomplete and local process knowledge which can be integrated dynamically at runtime, coupled with the fact that they are context-aware, makes pervasive process fragments a particularly suitable framework for pervasive applications.

Consider the following scenario inspired by airport checking and taxing procedures (Fig. 1). In this scenario, boxes containing goods arrive at the airport and go through customs before being released to their owners. Boxes will be treated and taxed differently depending on country of origin, content, etc. If a box contains forbidden items it will not be released to its owner, and will instead be disposed by authorized personnel.

The process knowledge for handling a box is distributed at different locations in the airport and depends on context information. Therefore, the complete flow model for treating a particular box is not known from the beginning. What is known is the goal of the flow: to release the box to its owner. The precise flow model that can achieve this goal is created at execution time, based on the available fragments.

We propose an approach for composing pervasive process fragments according to complex goals. Our approach is based on previous results from Web service composition [2], and in
particular on the work of [3], which addressed the problem of Web service composition with complex composition requirements. The problem of fragment composition is conceptually different from Web service composition, since the components are not orchestrated, but integrated into a complete flow model. Due to this difference, important issues in Web service composition, such as the need for Web services to communicate via message exchange, do not appear in fragment composition. On the other hand, fragment composition introduces new problems. For example, the fragments can overlap, or can have gaps in the specification. To deal with these differences, we have significantly modified and extended the approach in [3].

Our approach is based on a three-layer representation, where the first layer captures the specific domain knowledge required for the composition task, the second layer describes the abstract flow in terms of the goals it should achieve, and the third layer is the concrete, context-dependent part of the flow definition, represented using fragments.

The paper is structured as follows. In Section II, we describe our three-layer representation and the elements involved: entities, goals, and fragments. We describe a fragment composition technique based on planning in Section III, and evaluate it in Section IV. We position our work in a broader context in Section V and discuss future work in Section VI.

II. APPLICATION REPRESENTATION

For modeling an application, we consider that there exist three layers, arising from two distinctions (Fig. 2). First, we distinguish between domain knowledge, or knowledge about the entities in the domain, and process knowledge, or knowledge about business logic. Second, we distinguish between concrete, context-dependent knowledge and knowledge that is common, abstract, independent of context. The first layer is thus the domain knowledge, which is stable and abstract. The second layer is the stable part of the process knowledge, represented using goals, while the third is the dynamic and concrete part, represented using pervasive process fragments.

The domain knowledge consists of the types of entities in the domain. An entity type includes a set of properties (e.g., position, content) and a set of events that represent changes of these properties (e.g., if the content is checked it may be approved or rejected). Properties are states that last for a period of time, while events are actions that are effective at a certain point in time and make the entity evolve from one state to another.

We use the domain knowledge to define our goals. A goal can be used to model a flow or a part of a flow for which the exact content is not known at design-time. The exact content may remain unknown until the flow instance is executed. At that point, the goal will be substituted with a concrete realization, by composing available fragments. Using goals, we can specify the target state for our flow, as well as coordination requirements. A target state is a situation we want to achieve at the end of the flow execution, whereas a coordination requirement is a property we want to ensure during the entire execution.

To model entities and goals, we borrow the formalization proposed in [3]. In particular, we use object diagrams to represent entities, and a simple language for expressing goals with preferences for representing goals. Despite the different application domain, this formalization perfectly captures the core characteristics of our contextual entities and goals.

We use the domain knowledge also for annotating pervasive process fragments. We encode inside fragments the relation to one or more entities, and mark the points when the execution of the fragment would trigger also the evolution of the entities. Fragments consist of activities and control elements. To relate fragments to entities we use preconditions and effects, which can be defined on certain activities in the fragment. Preconditions are properties that have to hold for the activity to be applied in the composition. Effects are properties that are made true by applying the corresponding activity.

In this representation, both object diagrams and goals are specified independently from fragments. Therefore, using the same object diagrams and goals, we can achieve different fragment compositions for different contexts.

A. Object diagrams

An object diagram is a simple state transition system containing states which encode properties of an entity, and transitions between states triggered by events.

Definition 1 (Object Diagram): An object diagram representing an entity \( E \) is a tuple \( \langle L, L_0, \mathcal{E}, T \rangle \), where

- \( L \) is a finite set of configurations and \( L_0 \subseteq L \) is a set of initial configurations,
- \( \mathcal{E} \) is a set of events,
- \( T \subseteq L \times \mathcal{E} \times L \) is a transition relation that defines the evolution of the entity, based on events.

Fig. 3 displays the object diagrams in our scenario. Note that the tax invoice includes also the creation of the entity.

B. Goals

We express goals in terms of entities and their evolution. Goals can be used to specify desirable situations to be reached at the end of the composition, as well as rules to be maintained throughout the composition.

Definition 2 (Goal): A goal is defined with the generic constraint template \( \varphi \models \langle \varphi_1 \Rightarrow \ldots \Rightarrow \varphi_n \rangle \), where

\[ \varphi \equiv T \mid s^g(a) \mid e^g(a) \mid \varphi \lor \varphi \mid \varphi \land \varphi. \]
Here, \( s^o(o) \) defines the fact that diagram \( o \) is in configuration \( s \), and \( e^o(o) \) the fact that event \( e \) of \( o \) has taken place.

If the left side of the requirement is empty (\( \top \)), the rule specifies the need to unconditionally reach the state defined by the right side. Otherwise, the rule specifies that whenever the state in the left side occurs, the composition should try to reach the state defined in the right side. In both cases, the states on the right side are ordered using a preference operator (\( \succ \)), from the most preferred to the least preferred.

In our scenario, the goal is for box to reach the configuration \( READY \). If this is not possible, we at least want to have the box disposed of, therefore in configuration \( DISPOSED \): \( \top \Rightarrow ready^a(\text{box}) \succ disposed^a(\text{box}) \) \( (G_1) \)

C. Pervasive process fragments

Pervasive process fragments are the result of applying the process fragment definition from [1] to adaptable pervasive flows. They can be integrated into complete flow models by means of composition, and as such can be seen as the building blocks of flow models. In the following, we give a brief overview of adaptable pervasive flows and process fragments.

Adaptable pervasive flows are similar to the well-known workflows. They consist of activities and a corresponding execution order specified using control elements such as sequence, choice, parallel operators. Flows have associated constraints and address specific goals. We call flow instance a particular execution of a flow model. For describing flows we use a specialized language called Adaptable Pervasive Flow Language (APFL). The nucleus of APFL is BPEL [4], which has been extended to cover issues from the pervasive domain. APFL includes standard BPEL basic and structured activities (e.g., receive, reply, invoke, control constructs). It also includes APFL-specific activities defined as BPEL extensions (e.g., human interaction activity, context event). Further details on adaptable pervasive flows and APFL can be found in [5], [6].

Process fragments are a modeling approach which allows to model incomplete process knowledge. Here, there are three options. First, in a process fragment it is possible for control connectors to have either no source or no target activity. Second, the modeler has the freedom to not model control connectors at all. Third, process fragments can contain gaps, modeled using a special element called Region. A region helps to define an ordering between activities, when it is not clear what needs to happen in between.

D. Relating pervasive process fragments to object diagrams

We extend the fragment definition with relations to object diagrams: preconditions and effects. These can be attached to basic activities and to message handlers (onMessage, onHumanInteraction).

The preconditions (denoted with \( P_i \)) are propositional formulas over the set of propositions \( \{s^o(o_i)\} \), where \( o_i \) are
object diagrams and $s_j$ configurations. An activity annotated with a precondition requires the diagrams to be in particular configurations. If the precondition does not hold, the activity cannot be applied in the composition. For example, in the fragment **Check box origin**, activity **Check EU origin** requires box to be in configuration **UNLOADED**.

The effects (denoted with $E_i$) are sets of propositions from \{e_j^s(\omega_i)\}, where $\omega_i$ are object diagrams and $e_j$ events. An activity annotated with effects encodes the fact that diagrams may move to different configurations as a result of executing the activity. In the fragment **Check box content**, activity **Content nack** triggers the event **reject** on the diagram box. This can happen only if in the current configuration of box there exists a transition on **reject**, i.e., if box is in configuration **UNLOADED**. For readability, we have made this condition explicit using the precondition. However, such conditions can also be left implicit. From now on, by preconditions we will refer to the conjunction of explicit and implicit conditions.

**E. Overlapping activities**

A key issue about pervasive process fragments is that they can include overlapping activities. The reason is that fragment modelers have only a local view of the entire flow and may therefore model the same information.

Informally, two activities are overlapping if and only if there exists at least one object diagram for which they have the same effects, and their preconditions and effects are consistent. Two preconditions are consistent if they do not require any diagram to be in different configurations. Two effects are consistent if they do not trigger different transitions in the same diagram.

We introduce a helper formula $Xor$. For an object diagram $o = (L, L_0, E, T)$, $Xor(o)$ states that $o$ can be in exactly one configuration at a time:

\[
Xor(o) = (\bigvee_{s_i \in L} s_i^*(o)) \land \bigwedge_{s_i, s_j \in L, s_i \neq s_j} (\neg s_i^*(o) \lor \neg s_j^*(o))
\]

Let $a_1$ be an activity with preconditions $P_1$ and effects $E_1$ defined on a set of object diagrams $o_1, \ldots, o_k, o_{k+1}, \ldots, o_n$. Let $a_2$ be a second activity with preconditions $P_2$ and effects $E_2$ defined on $o_1, \ldots, o_k, o_{k+1}, \ldots, o_n$. We say that:

- $P_1$ and $P_2$ are consistent iff $P_1 \land P_2 \land \bigwedge_{1 \leq i \leq K} Xor(o_i)$ is satisfiable;
- $E_1$ and $E_2$ are consistent iff for all $o_i \in \{o_1, \ldots, o_k\}$, $E(o_i) = \{e_i^s(\omega_i) \land \neg e_j^s(\omega_i) \mid e_i^s(\omega_i) \in E \}$, we have $E_1 \cap E(o_i) = E_2 \cap E(o_i)$.

Further, $a_1$ and $a_2$ are overlapping iff:

- $k \geq 1$,
- $P_1$ and $P_2$, respectively $E_1$ and $E_2$, are consistent,
- there exists $o_i \in \{o_1, \ldots, o_k\}$, $o_i = (L, L_0, E, T)$, $E(o_i) = \{e_i^s(\omega_i) \mid e_i^s(\omega_i) \in E \}$, such that $E_1 \cap E(o_i) \neq \emptyset$.

Consider the activities **Charge tax** and **Charge tax with invoice** in the second and third fragment from Fig. 4. These activities are overlapping, even though they do not have the same name, preconditions, or effects. The activities have in common the object diagram box, and the consistency requirements are satisfied, since they both require box to be in configuration **EVALUATED** and trigger the same event **tax**.

**III. Solution**

The fragment composition problem can be stated as follows. Given a set of pervasive process fragments, a set of object diagrams, and a set of composition goals, the problem is to integrate a subset of the pervasive process fragments into an adaptable pervasive flow that achieves the goals.

Our solution is presented schematically in Fig. 5. First, we encode fragments, object diagrams and goals into a planning domain $\Sigma$ (steps 1-4). We then create the planning goal $\rho$ based on the goals given as input (step 5). On the domain $\Sigma$ and goal $\rho$ we apply the approach presented in [2], which generates a controller $\Sigma_c$ for controlling the domain $\Sigma$ in such a way as to satisfy the goal $\rho$ (step 6). In difference to [2], we are not interested to retrieve the controller $\Sigma_c$, but to analyze the controlled domain. If $\Sigma_c$ exists, then the controlled domain can be used to generate a synthesis of the fragments which achieves the composition goals (step 7).

The planning domain $\Sigma$ is defined as a state transition system (STS). An STS contains a set of states, of which some are marked as initial and/or final. Each state is labeled with sets of properties that hold in that state. The STS can evolve to new states as a result of performing actions. The actions can be either input (controllable) or output (not controllable).

**Definition 3 (STS):** Let $P$ be a set of proposition symbols and $Bool(P)$ the set of boolean expressions over $P$. A state transition system is a tuple $(S, S^0, I, O, R, S^F, F)$, where:

- $S$ is the set of states and $S^0 \subseteq S$ the set of initial states,
- $I$ and $O$ are the input and respectively output actions,
- $R \subseteq S \times Bool(P) \times (I \cup O) \times S$ is the transition relation,
- $S^F \subseteq S$ is the set of accepting states,
- $F : S \rightarrow 2^P$ is the labeling function.

In our planning domain $\Sigma$, the set $P$ consists of all propositions $\{s_j^s(\omega_i)\}$, encoding the fact that the diagram $\omega_i$ is in configuration $s_j$. The labeling function $F$ determines whether a boolean expression $b \in Bool(P)$ holds in a particular state $s$. We write $s, F \models b$ to denote that $b$ is satisfied at state $s$ given $F$. Satisfiability of a formula is determined according to the following standard inductive rules:
• $s, F \models \top$;
• $s, F \models p$, iff $p \in F(s)$;
• $s, F \models \neg b$, iff $s, F \not\models b$;
• $s, F \models b_1 \lor b_2$, iff $s, F \models b_1$ or $s, F \models b_2$.

The transitions in the STS are guarded: a transition $(s, b, a, s')$ is possible in state $s$ only if the guard expression $b$ is satisfied in that state, i.e., if $s, F \models b$.

To create our planning domain, we first transform fragments, object diagrams and goals into STSs. The planning domain will then be the parallel product of these STSs, capturing their simultaneous evolution. After building the planning domain and the planning goal, we determine if a controller exists such that the controlled domain satisfies the goal. We use the following notion of a controlled system.

Definition 4 (Controlled System): Let $\Sigma = (S, S^0, I, O, R, S^F, F)$ and $\Sigma_c = (S_c, S_c^0, I, O, R_c, S_c^F, F_c)$ be two STSs. STS $\Sigma_c \triangleright \Sigma$, describing the behaviors of system $\Sigma$ when controlled by $\Sigma_c$, is defined as follows:

$\Sigma_c \triangleright \Sigma = (S_c \times S, S_c^0 \times S^0, I, O, R_c \supset R, S_c^F \times S^F, F_c \cup F)$

where: $(s_c, s, b(a,b), a, (s'_c, s')) \in (R_c \circ R)$

if $(s_c, b_c, a, s'_c) \in R_c$ and $(s, b, a, s') \in R$.

A. Transforming the pervasive process fragments

To transform a fragment into an STS, we recursively translate its activities. Activity preconditions are copied as transition guards in the STS. We encode activities using input and output actions. Input actions correspond to activities that cannot be controlled by the composition, such as the onMessage within a pick. The idea is that once a pick is selected for composition, all its onMessage branches must be included. In this sense, it is not under the control of the composition whether to include the activity onMessage or not.$^1$

Table I contains the APFL elements and their translation to STS. To differentiate between input and output actions, we prepend the names with "?", respectively "!". We denote the recursive translation of activity $\alpha$ by $\Sigma_c \triangleright \Sigma$. Finally, we denote the boolean formula corresponding to the precondition of an activity with $\mathbf{b}$.

With this translation, we lose the information encoded in the effects of activities. We capture this information with a data structure called Action Table. The action table is used later on, when transforming object diagrams and goals. We add one entry in the action table for each occurrence of an activity with non-empty effects. An entry has the form $(b, a, \varepsilon)$, where $b$ and $\varepsilon$ are the precondition, respectively the effects of the activity, and $a$ is the action corresponding to the activity occurrence.

We capture the information regarding overlapping activities with a binary relation $\text{Overlap}$ defined on the set of actions.

$^1$Note that this encoding as input/output actions is different from the encoding of Web services in [3]. In Web service composition the purpose is not to integrate the Web services, but to orchestrate them using an external controller. In that setting, actions are considered to be input/output if they are controllable (non-controllable) by the external orchestrator.

<table>
<thead>
<tr>
<th>APFL Activity</th>
<th>STS Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic activity (receive, reply, invoke, contextEvent, etc.)</td>
<td>$\delta_{op} \rightarrow (s_0, b, \delta_{op}, s_1)$</td>
</tr>
<tr>
<td>sequence activity $a_1$</td>
<td>$\delta_{a_1} \rightarrow s_1$</td>
</tr>
<tr>
<td>activity $a_2$</td>
<td>$s_1 \xrightarrow{a_1} s'$, $s' \xrightarrow{a_2} s_2$</td>
</tr>
<tr>
<td>pick onMessage operations $\text{sup}_1$ activity $a_1$</td>
<td>$(s_0, T, \text{pick}, s_0)$</td>
</tr>
<tr>
<td>onMessage operations $\text{sup}_2$ activity $a_2$</td>
<td>$(s_0, b_1, \text{top}_2, s_1)$</td>
</tr>
<tr>
<td>activity $a_2$</td>
<td>$(s_0, b_2, \text{top}_2, s_2)$</td>
</tr>
<tr>
<td>apPick onHumanInteraction $\text{op}_1$ activity $a_1$</td>
<td>$(s_0, s_c, s_c \rightarrow s'_c)$</td>
</tr>
<tr>
<td>onHumanInteraction $\text{op}_2$ activity $a_2$</td>
<td>$(s_0, \text{switch}, s_0)$</td>
</tr>
<tr>
<td>switch case conditions: activity $a_1$</td>
<td>$(s_0, \text{if}, s_1), (s_0, \text{else}, s_2)$</td>
</tr>
<tr>
<td>otherwise activity $a_2$</td>
<td>$(s_0, \text{otherwise}, s_2)$</td>
</tr>
<tr>
<td>*here, $c \in \text{Bool}({s_j(a)})$</td>
<td>$s_1 \xrightarrow{c} s_0, s_2 \xrightarrow{\neg c} s'_c$</td>
</tr>
<tr>
<td>flow activity $a_1$ activity $a_2$</td>
<td>$(s_0, T, \text{flow}, s_0)$</td>
</tr>
<tr>
<td>$(s_0, T, \text{order}, s_1)$</td>
<td>$(s_0, T, \text{order}, s_2)$</td>
</tr>
<tr>
<td>$(s_0, T, \text{order}, s_2)$</td>
<td>$(s_1, s_0, s_1 \rightarrow s_0)$</td>
</tr>
<tr>
<td>$(s_1, s_1 \rightarrow s_1)$</td>
<td>$(s_0, s_1 \rightarrow s_1)$</td>
</tr>
<tr>
<td>region</td>
<td>$(s_0, T, \text{region}, s_0)$</td>
</tr>
<tr>
<td>$(s_0, T, \text{region}, s_1)$</td>
<td>$(s_0, T, \text{region}, s_2)$</td>
</tr>
</tbody>
</table>

TABLE I: Translating APFL to STS

For every pair $(a_1, a_2) \in \text{Overlap}$, $a_1$ and $a_2$ are actions corresponding to two overlapping activities. We use the Overlap relation also for encoding the fact that regions are substitutes for one or more activities. If $a_r$ is an action corresponding to a region in the fragment $F$, we add Overlap relations between $a_r$ and every action $a$ that corresponds to a fragment $F' \neq F$.

Fragments can be used also partially, and therefore by default all states are accepting (i.e., $S^F = S$). However, when a fragment includes the structured activity flow, we remove from $S^F$ all the states included in the translation of the activity, except for the initial and final state.

The translation of the fragment Bring to claim from Fig. 4 is an STS $\Sigma = (S, S^0, I, O, R, S^F, F)$, where:

- $S = \{s_0, s_1, s_2\}$
- $S^0 = \{s_0\}$
- $I = \{F_5.\text{Take_to_baggage_claim}, F_5.\text{Box_at_baggage_claim}\}$
- $O = \emptyset$
- $R$ contains the following transitions:
  - $(s_0, \text{approved}(\text{box}), F_5.\text{Take_to_baggage_claim}, s_1)$
  - $(s_1, \text{approved}(\text{box}), F_5.\text{Box_at_baggage_claim}, s_1)$
- $S^F = \{s_0, s_1, s_2\}$
- $F = \emptyset$

For this fragment, we also add one action table entry:

$\text{approved}(\text{box}), F_5.\text{Box_at_baggage_claim}, \{\text{release}(\text{box})\}$

B. Transforming the object diagrams

Given an object diagram $o = (L, L_0, E, T)$, we define an STS $\langle S, S^0, I, O, R, S^F, F \rangle$, where $S = L$, $S^0 = L_0$, and all
states are accepting ($S^F = S$). Further, we label states with the corresponding propositions (i.e., $\forall s \in S : F(s) = \{s^d(o)\}$).

We build the transitions in the new STS using the action table. For each transition $(l, c, l') \in T$, we consider all the entries $(b, a, \varepsilon)$ in the action table, such that $e^b(o) \in \varepsilon$. For each such entry $(b, a, \varepsilon)$, we add to $R$ the transition $(l, b, a, l')$.

For example, in the STS of box, we use the original transition (APPROVED, release, READY) and the entry (approved(box), ¿F5.Box_at_baggage_claim, {release(e(box))}) to create the transition (APPROVED, approved(box), ¿F5.Box_at_baggage_claim, READY).

C. Transforming the goals

For each goal, we construct the STSs that correspond to the satisfiability of the goal. For every formula $\varphi$, we define a single output action $e_\varphi$ which gets triggered when the formula is satisfied. We use these completion actions for composing the formulas. The preconditions on the activities will be carried over as guards also in the goal STSs. We shortly describe the STSs for the building blocks of goal formulas:

- the STS for $\top$ has one transition on the completion action;
- the STS for $s^d(o)$ has one transition guarded with the corresponding proposition;
- the STS for $e^b(o)$ waits for any activity that contains the event in its effects. For each action entry $(b, a, \varepsilon)$ such that $e^b(o) \in \varepsilon$, the STS has transition on $a$ guarded by $b$.
- The STS for $\varphi_1 \lor \varphi_2$ waits for any of $e_{\varphi_1}$ and $e_{\varphi_2}$, while the STS for $\varphi_1 \land \varphi_2$ waits for both.

The STS for a goal formula $\varphi \implies (\varphi_1 \implies \ldots \implies \varphi_n)$ is as follows. If the premise takes place ($e_\varphi$ is reported), it moves to a non-accepting state and waits for any of $e_{\varphi_1}, \ldots, e_{\varphi_n}$ to be reported. The goal with preferences has the form: $\rho_\varphi = (s_0, s_1, \ldots, s_n)$, where $s_0$ is the initial state of the STS, and $s_1, \ldots, s_n$ are the states reached with the transitions corresponding to $e_{\varphi_1}, \ldots, e_{\varphi_n}$.

Fig. 6 presents the STSs for the goal $G_1$ introduced in Section II-B. The goal with preferences is $\rho_\varphi = (s_0, s_1, s_2)$.

D. Generating the composed adaptable pervasive flow

We build the planning domain by taking the parallel product of all the STSs of fragments, object diagrams and goals.

One key idea here is to enforce actions to be applied in parallel if they overlap and their preconditions hold. Slightly abusing the notation, we write $a_1||a_2$ to denote a new action which corresponds to actions $a_1$ and $a_2$ being performed in parallel. We call $a||a'$ a parallel action. Note that $a$ and $a'$ can themselves be parallel actions.

The overlap relation can now be extended to cover parallel actions. Given two actions $a \equiv a_1 \ldots \equiv a_n$ and $a' \equiv a'_1 \ldots \equiv a'_m$, if for all $1 \leq i \leq n$ and $1 \leq j \leq m$, we have $(a_i, a'_j) \in \text{Overlap}$, then also $(a, a') \in \text{Overlap}$.

We use a simplifying observation when constructing the parallel product: that an action always appears with the same guard. By construction, each action corresponds to at most one occurrence of an activity in a fragment. In case the action corresponds to an activity, the guard is the precondition of this activity. Otherwise, the guard is equal to $\top$. The action-guard relation is maintained when transforming object diagrams and goals, since we always add the action together with the guard.

Definition 5 (Parallel Product): Let $\Sigma_1 = \langle S_1, S_1^F, I_1, O_1, R_1, S_1^E, F_1 \rangle$ and $\Sigma_2 = \langle S_2, S_2^F, I_2, O_2, R_2, S_2^E, F_2 \rangle$ be two STSs. Let $\text{Overlap}$ be a binary relation. The parallel product $\Sigma_1 \parallel \Sigma_2$ is defined as $\langle S_1 \times S_2, S_1^E \times S_2^E, I_1 \cup I_2, O_1 \cup O_2, R_1 \parallel R_2, S_1^F \times S_2^F, F_1 \parallel F_2 \rangle$.

where $(F_1 \parallel F_2)(s_1, s_2) = F_1(s_1) \cup F_2(s_2)$ and for every state $(s_1, s_2) \in S_1 \times S_2$:

- if $(s_1, b, a, s'_2) \in R_1 \text{ and } (s_2, b, a, s_2') \in R_2$, then $(\langle s_1, s_2 \rangle, b, a, (s_1, s_2')) \in R_1 \parallel R_2$;
- if $(s_1, b_1, a_1, s'_1) \in R_1 \text{ and } \forall (s_2, b_2, a_2, s_2') \in R_2, a_1 \neq a_2 \text{ and } (a_1, a_2) \notin \text{Overlap}$, then $(\langle s_1, s_2 \rangle, b_1, a_1, (s_1, s_2')) \in R_1 \parallel R_2$;
- if $(s_1, b, a, s_2') \in R_2 \text{ and } \forall (s_1, b_1, a_1, s'_1) \in R_1, a_1 \neq a_2 \text{ and } (a_1, a_2) \notin \text{Overlap}$, then $(\langle s_1, s_2 \rangle, b, a_2, (s_1, s_2')) \in R_1 \parallel R_2$;
- otherwise, if $(s_1, b_1, a_1, s'_1) \in R_1 \text{ and } (s_2, b_2, a_2, s'_2) \in R_2 \text{ and } (a_1, a_2) \in \text{Overlap}, (R_1 \parallel R_2)$ contains:

  - $(\langle s_1, s_2 \rangle, b_1 \land b_2, a_2, (s_1, s_2'))$;
  - $(\langle s_1, s_2 \rangle, b_1 \land \neg b_2, a_1, (s_1, s_2'))$;
  - $(\langle s_1, s_2 \rangle, \neg b_1 \land b_2, a_2, (s_1, s_2'))$.

We construct our planning domain $\Sigma$ as the parallel product of fragment STSs $\Sigma_{F_1}, \ldots, \Sigma_{F_n}$, object diagram STSs $\Sigma_{O_1}, \ldots, \Sigma_{O_m}$, and goal STSs $\Sigma_{C_1}, \ldots, \Sigma_{C_k}$:

$\Sigma = \Sigma_{F_1} \parallel \ldots \parallel \Sigma_{F_n} \parallel \Sigma_{O_1} \parallel \ldots \parallel \Sigma_{O_m} \parallel \Sigma_{C_1} \parallel \ldots \parallel \Sigma_{C_k}$.

We simplify $\Sigma$ by removing the transitions $(s, b, a, s') \in R'$ for which $a$ corresponds to a region. Let $R' \subseteq R$ be the updated transition relation. This condition comes from the fact that region actions should not be applied independently, since we are interested in obtaining a region-free composition.

Further, we remove all transitions $(s, b, a, s') \in R'$ which can never fire, i.e., for which $s, F \not\models b$. On the resulting STS, we can then remove the guards as well as the labeling function. The simplified planning domain, which we denote with $\Sigma_s$, is a tuple $\langle S', S^0, I, O, R''', S''' \rangle$, where $R''' \subseteq S \times (I \cup O) \times S$ is the transition relation after the updates.

We then construct the planning goal $p$ by combining the composition goals: $p = \land_\rho \rho_\varphi$.

Given the domain $\Sigma_1$ and the goal $\rho$, we apply the technique presented in [2], which generates a controller $\Sigma_c$ such that $\Sigma_c \parallel \Sigma_s \models p$. If such a controller exists, then the controlled domain $\Sigma_c \parallel \Sigma_s$ corresponds to a synthesis of the fragments which achieves the composition goals.

We obtain the composed flow model by translating to APFL the STS of the controlled domain, to which we add potentially
missing connectors and start activities. The translation to
APFL is conceptually simple: from the construction of STSs,
each action name is unique and corresponds to at most one
appearance of an activity in a fragment. Such actions can be
mapped back to their corresponding activities. In the case of
parallel actions, we introduce new activities, which result from
merging the original overlapping activities.

Fig. 7 presents the result of composition in our scenario.
Note that the scenario has no solution for any of the separate
goals $\top \Rightarrow \text{ready}^s(\text{box})$, respectively $\top \Rightarrow \text{disposed}^s(\text{box})$.

IV. EValuation

We implemented our approach into a prototype tool. The
tool translates object diagrams (XML), fragments and goals
(APFL) into a planning problem. The planning problem is
given as input to WSYNTH, one of the tools in the ASTRO
toolset (http://www.astroproject.org). The output returned by
WSYNTH is the controlled domain $\Sigma_c \triangleright \Sigma_s$, which we then
translate back to APFL.

We evaluated our tool using a dual-core CPU running at
2.26GHz, with 3GB memory. For each experiment, we report
the averages over 20 runs. The tool takes 0.58 seconds to solve
our Box at the airport scenario, with the fragments from Fig. 4.

We consider two features specific to fragment composition.
First, there is a tradeoff between designing fragments with a
large number of activities (a higher burden on the designer)
and with a small number (longer composition time). Second,
the set of available fragments can contain more fragments then
actually necessary for composition.

In the first experiment, we evaluate the impact of fragment
sizes on performance. Intuitively, larger fragments should be
especially useful for hard composition problems. We therefore
consider the following scenario. Assume that a group of $k$
people is traveling to a conference. The conference has events
that overlap in time, with up to $k$ events running concurrently.
The goal is to ensure that every event is attended by one
person. This scenario is equivalent to a graph $k$-coloring
problem, which is NP-complete for $k \geq 3$. We encode the
scenario for $k = 3$. Activities assign events to people, with
the time overlap constraints represented as preconditions. At
each step, we increase the number of events and measure
the performance for fragments of different sizes, starting
with fragments of one. For this purpose, we randomly group
activities into semantically correct fragments. Note that only a
third of the fragments are actually composed. Fig. 8a presents
the composition time for up to 36 activities. We observe that
larger fragments lead to a significant speedup. Using fragments
of two or more activities can lead to a time increase of
20% in the worst case, and a decrease of up to 95% in the
best case. For this domain involving complex composition
problems, larger fragments provide a clear benefit in terms
of composition time.

In the second experiment, we test how the number of avail-
able fragments influences the performance. For this purpose,
we vary the total number of fragments, while keeping constant
the number of fragments actually composed. We use our
Box at the airport scenario. At each step, we generate new
fragments as copies of random fragments from our original set.
Fig. 8b presents the composition time for up to 28 fragments.
As expected, the composition time grows exponentially in the
number of fragments, due to the hardness of the planning
problem. However, it is a promising result that the composition
takes a reasonable time for a fairly large number of fragments,
since we can use existing selection techniques to reduce this.

V. RELATED WORK

All business processes are designed to achieve a business
goal. The goal can be represented using either an imperative or
a declarative language. Imperative process models (e.g., [4])
focus on how: they describe the way to achieve the goal,
and assume that the environment is stable. Declarative process
models (e.g., [7]) focus on what: they specify the goal through
constraints which approximate the desired behavior. Such
descriptions are suitable for frequently changing environments,
but cannot be executed completely automatically.

We use both process description techniques to achieve
a greater flexibility with respect to the environment. The
declarative part corresponds to goals, and is globally available.
The imperative, concrete part corresponds to pervasive process
fragments, and is available only locally. Another approach
combining the two techniques is [8], which introduces a
framework for defining semantic constraints over processes.
If the imperative definition of a process instance is changed
during execution, the framework allows to check whether it is
still compliant to the declarative, semantic definition.

Fragment composition is also an implementation of process
flexibility, as classified in [9]. Our approach supports flexibility
by underspecification, when the process model contains place-
holders for which a concrete realization is provided at runtime.
This realization is selected in case of late binding (e.g., [10],
[11]), and constructed for late modeling (e.g., [12]). In our
approach the placeholder is the goal, and both late binding and
late modeling can be achieved. Differently from our approach, [10] and [11] do not compose fragments, they select and dynamically bind complete process models. In [12], instances are progressively built during execution, based on constraints which specify how and when the fragments can be composed. Such concrete constraints are absent in our approach.

[13]–[15] are close to our work from a modeling perspective. In [13], the authors model small processes as ‘proclets’, a modeling metaphor similar to process fragments, based on the same assumption that process knowledge is distributed. Unlike fragments, proclets get integrated into a business process by interaction. In [14], process models are generated starting from object life cycles. While [14] uses object life cycle conformance and coverage requirements, we use the more expressive goals with preferences. [15] discusses the artifact-centric paradigm and its dimensions: (i) business artifacts, (ii) artifact lifecycles, (iii) services/tasks which make transactional changes to artifacts, and (iv) constraints on these changes. Our approach can also be seen as artifact-centric, with object diagrams, effects, and preconditions corresponding to (ii)-(iv).

Closely related to fragment composition is the problem of Web service composition, that of generating a composite service starting from service interfaces and composition requirements. The result is an executable implementation which satisfies the requirements by suitably invoking the existing Web services. This is different from fragment composition, where fragments are integrated into a new flow model. ASTRO (see [2], [3]) is a Web service composition approach, which is also the starting point of our work. In particular, we have exploited the goal language and object diagrams introduced in [3] to model our application, and used the powerful planning techniques in ASTRO to implement our solution.

VI. CONCLUSIONS

We presented an approach for composing pervasive process fragments according to context and goals. We first specified the properties of domain entities using object diagrams, and used these properties to define our goals and to annotate the fragments. We then encoded object diagrams, goals and fragments into an AI planning problem. The result of composition is a complete adaptable pervasive flow which satisfies the goals and corresponds to a synthesis of the fragments.

In our future work, we will address several open issues. First, we plan to extend our composition mechanism with adaptation constructs provided by APFL, such as the abstract activity (described in terms of goals) and the built-in adaptation tools [5]. Further, we plan to evaluate our approach on a real-world scenario developed in the project ALLOW, which involves vehicle logistics. As a last issue, until now we have considered only the lifecycle aspect of the data, in the form of object diagrams. We plan to extend our object diagrams and fragments to include also an information model.

Acknowledgment: This work is partially funded by the ALLOW project. ALLOW is part of the EU 7th Framework Programme (contract no. FP7-213339).

REFERENCES