Synthesizing a Library of Process Templates through Partial-Order Planning Algorithms

Andrea Marrella[1] and Yves Lespérance[2]

[1] Sapienza - Universitá di Roma, Italy
[2] York University, Toronto

Abstract. The design time specification of dynamic processes can be time-consuming and error-prone, due to the high number of tasks involved and their context-dependent nature. Such processes frequently suffer from potential interference among their constituents, since resources are usually shared by the process participants and it is difficult to foresee all the potential tasks interactions in advance. Concurrent tasks may not be independent from each other (e.g., they could operate on the same data at the same time), resulting in incorrect outcomes. To address these issues, we propose an approach that exploits partial-order planning algorithms for automatically synthesizing a library of process template definitions for different contextual cases. The resulting templates guarantee sound concurrency in the execution of their activities and are reusable in a variety of partially-known contextual environments.

1 Introduction

Current workflow technology is based on the idea that there *always* exists an underlying fixed process that can be used to automate the work [1]. Once identified, a process is formalized into a *process model* which captures every possible case (i.e., *process instance*) to be executed at run-time through a Process Management System (PMS). This approach works for processes where procedures are well known, repeatable and can be planned in advance with some level of detail. In recent years, the need to deal with *dynamic processes* and provide support for flexible process management has emerged as a leading research topic in the Business Process Management (BPM) domain [2]. In a dynamic process, the sequence of tasks depends heavily on the specifics of the context (e.g., which resources are available and what particular options exist at the time), and it is often unpredictable how the process will unfold. The design-time specification of all possible cases requires an extensive manual effort for the process designer, who has to anticipate all potential alternatives into the process model, in an attempt to deal with the context dependent nature of these processes (cf. Section 2). Such processes do not have the same level of repeatability of classical business processes, and the execution changes on a case-by-case basis, generating instances that are different almost every time, depending on the context.

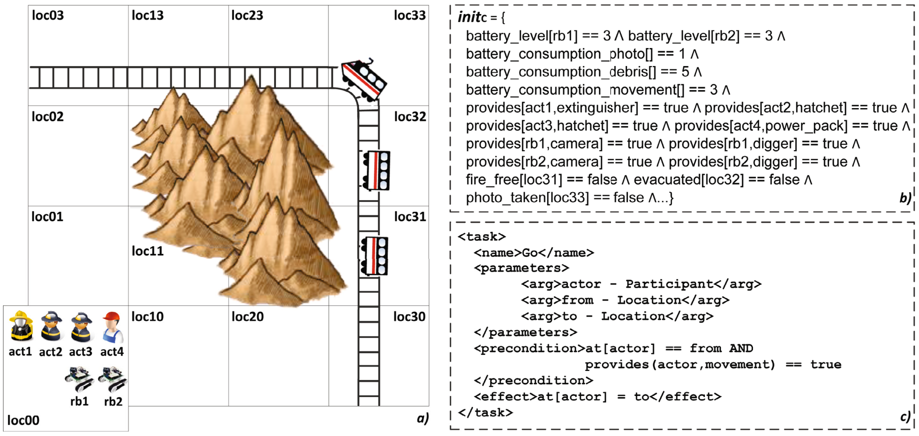
In this paper, we present an approach that allows us to automatically synthesize a *library* of *process templates* starting from a representation of the contextual domain in which the process is embedded in and from an extensive repertoire of tasks defined for such a context. A template depicts the best-practice procedure drawn up with whatever contextual information available at the time; it describes a recommended control flow for the process that can be enacted in a range of states satisfying the context conditions. In order to build process templates, we make use of *partial-order planning algorithms* (aka POP [3]), which guarantee some interesting properties in the construction of the template:

- *Sound concurrency.* A template has the property of *sound concurrency* in the execution of its concurrent activities, that are proven to be effectively *independent* one from another (i.e., at runtime there is no risk of interference between concurrent tasks, since they cannot affect the same data).
- *Executability in partially known environments.* Once synthesized, a template can be executed in several starting states, since it (usually) requires a fragment of the knowledge of the starting state to successfully achieve its objectives. We identify the *weakest preconditions* of process templates, and all the states satisfying such preconditions are good candidates for executing them.

We exploit the idea behind POP of representing flexible plans that enables deferring decisions. Instead of committing prematurely to a complete, totally ordered sequence of actions, plans are represented as a partially ordered set, and only the required ordering decisions are recorded. A process template is generated on the basis of such a set of activities, and we are able to identify what knowledge about the starting state is required for successful template execution. Moreover, we build step-by-step a library of process template specifications and support efficient retrieval of appropriate templates in partially known environments.

2 A Running Example

Let us consider the emergency management scenario described in Fig. 1(a). It concerns a train derailment and depicts a map of the area (as a 4x4 grid of locations) where the disaster happened. We suppose that the train is composed of a locomotive (located in *loc33*) and two coaches (located in *loc32* and *loc31* respectively). The goal of an incident response plan defined for such a context is to evacuate people from the coach located in *loc32*, to extinguish a fire in the coach in *loc31* and finally to take pictures for evaluating possible damages to the locomotive, located in *loc33*. Thus, a response team can be sent to the derailment scene. The team is composed of four first responders (in the remainder, we refer to them as *actors*) and two robots, initially located in *loc00*. We assume that actors are equipped with mobile devices (for picking up and executing tasks) and provide specific capabilities. For example, actor *act1* is able to extinguish fires, while *act2* and *act3* can evacuate people from train coaches. The two robots, instead, may take pictures and remove debris from specific locations. Each robot has a battery and each action consumes a given amount of battery charge. When the battery of a robot is discharged, actor *act4* can charge it. Fig. 1(b) summarizes the above.



```
initc = {
 battery_level[rb1] == 3 ∧ battery_level[rb2] == 3 ∧
 battery_consumption_photo[] == 1 ∧
 battery_consumption_debris[] == 5 ∧
 battery_consumption_movement[] == 3 ∧
 provides[act1,extinguisher] == true ∧ provides[act2,hatchet] == true ∧
 provides[act3,hatchet] == true ∧ provides[act4,power_pack] == true ∧
 provides[rb1,camera] == true ∧ provides[rb1,digger] == true ∧
 provides[rb2,camera] == true ∧ provides[rb2,digger] == true ∧
 fire_free[loc31] == false ∧ evacuated[loc32] == false ∧
 photo_taken[loc33] == false ∧...}
```

```
<task>
  <name>Go</name>
  <parameters>
        <arg>actor - Participant</arg>
        <arg>from - Location</arg>
        <arg>to - Location</arg>
  </parameters>
  <precondition>at[actor] == from AND
                provides(actor,movement) == true
  </precondition>
  <effect>at[actor] = to</effect>
</task>
```

Fig. 1. Area and context of the intervention

The definition of an incident response plan as a business process involves a dynamically selected set of activities to be executed on the field by the first responders. Since the process may be different every time it is defined because it strictly depends on the actual contextual information (the positions of actors/robots, the battery level of robots, etc.), it is unrealistic to assume that the process designer can pre-define all the possible process models for dealing with this environment (apparently simple). Moreover, if contextual data describing the environment are known, the synthesis of a process dealing with such an environment is not straightforward, as the correctness of the process model is constrained by the values (or combination of values) of contextual data. A simple approach to solving our problem is to build a process as a sequence of activities, e.g., the sequence of actions shown in Fig. 2. However, this solution is highly "inefficient", as many actions are independent, and they could be executed concurrently to reduce intervention time; e.g., a robot could take pictures in parallel with the extinguishing of the fire in *loc31*. But, at the same time, a process designer may find it difficult to organize activities for concurrent execution, since each action, for its executability, depends on the values of contextual data (e.g., a robot needs enough battery charge for moving into a location and taking pictures or removing debris). Also dependencies between actions play a key role in the definition of the process model (e.g., in order to evacuate people at *loc32*, a robot must have removed the debris beforehand). Finally, a process designer tends to represent more contextual information than that strictly needed for defining a process. For example, the process in Fig. 2 does not involve actor *act3*, meaning that any information concerning *act3* (e.g., its capabilities, its location, etc.) is not required for synthesizing and executing the process. To overcome the above issues, we propose a solution that involves exploiting *partial-order planning* for generating a library of *process templates* for different contextual cases. Our templates provide *sound concurrency* in the execution of their activities and are executable in *partially known environments*.

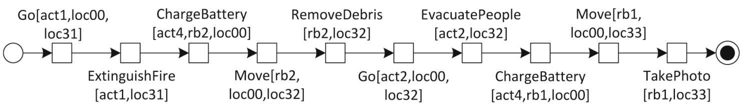


Fig. 2. A process dealing with the scenario of Fig. 1

3 Partial-Order Planning

Planning systems are problem-solving algorithms that operate on explicit representations of states and actions. The standard representation language for classical planners is known as the Planning Domain Definition Language (PDDL [4]); it allows us to formulate a problem PR through a set of possible actions, the description of the initial state of the world $init_{\mathsf{PR}}$ and of the desired goal condition $goal_{\mathsf{PR}}$. The set of all action definitions Ω is the *domain* PD of the planning problem. Each action $a \in \Omega$ has a preconditions list (stating the atomic conditions under which an action can be executed) and an effects list to be applied on the state of the world, denoted respectively as Pre_a and Eff_a. A planner that works on such inputs generates a sequence of actions (the *plan*) that corresponds to a path from the initial state to a state meeting the goal condition.

In this paper, we focus on *Partial-Order Planning (POP)* [3], a specific kind of plan-space search algorithm. A plan space is an implicit directed graph whose nodes are *partially specified plans* and whose edges correspond to refinement operations that further complete a partial plan, i.e., to achieve an open goal or to remove possible inconsistencies. POP takes as input a PDDL planning problem and searches the space of partial plans without committing to a totally ordered sequence of actions. Basically, a **partial plan** is a tuple $\mathsf{P} = (A, O, CL)$, where $A \subseteq \Omega$ is a set of (ground) actions, O is a set of *ordering constraints* over A, and CL is a set of *causal links* over A. Ordering constraints O are of the form $a \prec b$, which is read as "a before b" and means that action a must be executed sometime before action b, but not necessarily immediately before. Causal links CL may be represented as $c \xrightarrow{p} d$, which is read as "c achieves p for d" and means that p is an effect of action c and a precondition for action d. A precondition without a causal link requires further refinement to the plan to establish it, and is considered to be an *open condition* in the partial plan. A classical POP algorithm starts with a null partial plan P and keeps refining it until a solution plan is found. The null partial plan contains two dummy actions $a_0 \prec a_\infty$ where the preconditions of a_∞ correspond to the top level goals $goal_{\mathsf{PR}}$ of the problem, and the effects of a_0 correspond to the conditions in $init_{\mathsf{PR}}$. Intuitively, a refinement operation avoids adding to the partial plan any constraints that are not strictly needed for addressing the refinement objective. This is called the *least commitment principle* [3], and its advantage is that decisions about action ordering are postponed until a decision is forced; constraints are not added to a partial plan unless strictly needed, thus guaranteing flexibility in the execution of the plan and by allowing actions to run concurrently. A **consistent** plan is defined as a plan with no cycles in the ordering constraints and no conflicts with the causal links. A consistent plan with no open conditions is a **solution** [3].

4 Process Templates

The synthesis of a dynamic process requires a tight integration of process activities and contextual data in which the process is embedded in. The context is represented in the form of a *Domain Theory* D, that captures a set of tasks $t_i \in \mathsf{T}$ (with $i \in 1..n$) and supporting information, such as the people/agents that may be involved in performing the process (roles or participants), the data and so forth. Tasks are collected in a specific repository, and each task can be considered as a single step that consumes input data and produces output data. Data are represented through some ground atomic terms $v_1[y_1], v_2[y_2], ..., v_m[y_m] \in \mathsf{V}$ that range over a set of tuples (i.e., unordered sets of zero or more attributes) $y_1, y_2, \dots y_m$ of *data objects*, defined over some *data types*. In short, a data object depicts an entity of interest; for example, in our scenario we need to define data objects for representing participants (e.g., data type $Participant = \{act1, act2, act3, act4, rb1, rb2\}$), capabilities (e.g., data type $Capability = \{extinguisher, movement, \dots hatchet\}$) and locations in the area (e.g., data type $Location = \{loc00, loc10, \dots loc33\}$). Each tuple y_j may contain one or more data objects belonging to different data types. The domain $dom(v_j[y_j])$ over which a term is interpreted can be of various types: (i) *Boolean*: $dom(v_j[y_j]) = \{true, false\}$, (ii) *Integer*: $dom(v_j[y_j]) = \mathbb{Z}$, (iii) *Functional*: the domain contains a fixed number of data objects of a designated type. Terms can be used to express properties of domain objects (and relations over objects). In our example, we may need boolean terms for expressing the presence of a fire in a location (e.g., $fire\_free[loc : Location] = (bool : Boolean)$), integer terms for representing the battery charge level of each robot (e.g., $battery\_level[prt : Participant] \in \mathbb{Z}$) or functional terms for recording the position of each actor in the area (e.g., $at[prt : Participant] = (loc : Location)$). Moreover, since each task has to be assigned to a participant that provides all of the skills required for executing that task, there is the need to consider the participants "capabilities". This can be done through a boolean term $provides[prt : Participant, cap : Capability]$ that is $true$ if the capability cap is provided by prt and $false$ otherwise.

Each task is annotated with *preconditions* and *effects*. Preconditions can be used to constrain the task assignment and must be satisfied before the task is applied, while effects establish the outcome of a task after its execution. Note that, as shown in Fig. 3(a), our approach treats each task as a "black box" and no assumption is made about its internal behavior (we consider the task execution as an instantaneous activity).

Definition 1. *A* task $t[x] \in \mathsf{T}$ *consists of:*

- *a tuple of data objects x as input parameters;*
- *a set of preconditions Pre_t, represented as the conjunction of k atomic conditions defined over some specific terms, $Pre_t = \bigwedge_{l \in 1..k} pre_{t_l}$. Each pre_{t_l} can be represented as $\{v_j[y_j]$ **op expr**$\}$, where:*
  - *$v_j[y_j] \in \mathsf{V}$ is an atomic term, with $y_j \subseteq x$, i.e., admissible data objects for y_j need to be defined as task input parameters;*

  - *An **expr** can be a <u>boolean value</u> (if v_j is a boolean term); an <u>input parameter</u> identified by a data object (if v_j is a functional term); an <u>integer number</u> or an <u>expression</u> involving integer numbers and/or terms, combined with the arithmetic operators {+,-} (if v_j is a integer term);*
  - ***op*** $\in \{<, >, ==, \leq, \geq\}$ *is a relational operator.*
- *a set of deterministic effects Eff_t, represented as the conjunction of h atomic conditions defined over some specific terms, $Eff_t = \bigwedge_{l \in 1..h} eff_{t_l}$. Each eff_{t_l} (with $l \in 1..h$) can be represented as $\{v_j[y_j]$ **op expr**$\}$, where:*
  - *$v_j[y_j] \in \mathsf{V}$ and **expr** are defined as for preconditions.*
  - ***op*** *$\in$ {=,+=,-=} is used for assigning (=) to a term a value consistent with the **expr** field or for incrementing (+ =) or decrementing (− =) an integer term by that value.*

For example, the task *Go* described in Fig. 1(c) involves two parameters *from* and *to* of type *Location* and a parameter *actor* of type *Participant*. An instance of *Go* can be executed only if *actor* is currently at the starting location *from* and provides the required capabilities for executing the task. As a consequence of task execution, the actor moves from the starting to the arrival location, and this is reflected by assigning to the term $at[actor]$ the value *to* in the effect.

Modeling a business process involves representing how a business pursues its objectives/goals. The goal may vary depending on the specific *Process Case* C to be handled. A case C reflects an instantiation of the domain theory D with a starting condition $init_{\mathsf{C}}$ and a goal condition $goal_{\mathsf{C}}$. Both conditions are conjunctions of atomic terms. We do not assume complete information about $init_{\mathsf{C}}$; this means we allow a process designer to instantiate only the atomic terms necessary for representing what is known about the starting state, i.e., $init_{\mathsf{C}} = \{v_1[y_1] == val_1 \wedge ... \wedge v_j[y_j] == val_j\}$, where val_j (with $j \in 1..m$) represents the j-th value assigned to the j-th atomic term. Fig. 1(b) shows a portion of $init_{\mathsf{C}}$ concerning the scenario depicted in Fig. 1(a). The goal is a condition represented as a conjunction of some specific terms we want to make true through the execution of the process. For example, in the scenario shown in Section 2, the goal has to be represented as : $goal_{\mathsf{C}} = \{fire\_free[loc31] == true \wedge evacuated[loc32] == true \wedge\ photo\_taken[loc33] == true\}$. The syntax of goal conditions is the same as for tasks preconditions.

A state is a complete assignment of values to atomic terms in V. Given a case C, an intermediate state $state_{\mathsf{C}_i}$ is the result of i tasks performed so far, and atomic terms in V may be thought of as "properties" of the world whose values may vary across states.

Definition 2. *A task t can be performed in a given $state_{\mathsf{C}_i}$ (and in this case we say that t is **executable** in $state_{\mathsf{C}_i}$) iff $state_{\mathsf{C}_i} \vdash Pre_t$, i.e. $state_{\mathsf{C}_i}$ **satisfies** the preconditions Pre_t for the task t.*

Moreover, if executed, the effects Eff_t of t modify some atomic terms in V and change $state_{\mathsf{C}_i}$ into a new state $state_{\mathsf{C}_{i+1}} = update(state_{\mathsf{C}_i}, Eff_t)$. The *update* function returns the new state obtained by applying effects Eff_t on the current state $state_{\mathsf{C}_i}$. Starting from a domain theory D, a *Process Template*

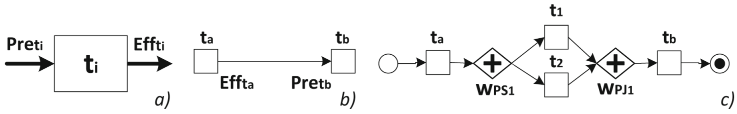


Fig. 3. Task anatomy (a), causality (b) and concurrency (c) in a process model

captures a partially ordered set of tasks, whose successful execution (i.e., without exceptions) leads from $init_{\mathsf{C}}$ to $goal_{\mathsf{C}}$. Formally, we define a template as a directed graph consisting of tasks, gateways, events and transitions between them.

Definition 3. *Given a domain theory* D*, a set of tasks* T *and a case* C*, a Process Template* PT *is a tuple (N,L) where:*

- $N = T \cup E \cup W$ *is a finite set of nodes, such that :*
  - T *is a set of tasks instances, i.e., occurrences of a specific task* $t \in \mathsf{T}$ *in the range of the process template;*
  - E *is a finite set of events, that consists of a single start event* $\bigcirc$ *and a single end event* $\odot$*;*
  - $W = W_{PS} \cup W_{PJ}$ *is a finite set of parallel gateways, represented in the control flow with the* $\diamond$ *shape with a "plus" marker inside.*
- $L = L_T \cup L_E \cup L_{W_{PS}} \cup L_{W_{PJ}}$ *is a finite set of transitions connecting events, task instances and gateways:*
  - $L_T : T \to (T \cup W_{PS} \cup W_{PJ} \cup \odot)$
  - $L_E : \bigcirc \to (T \cup W_{PS} \cup \odot)$
  - $L_{W_{PS}} : W_{PS} \to 2^T$
  - $L_{W_{PJ}} : W_{PJ} \to (T \cup W_{PS} \cup \odot)$

The constructs used for defining a template are essentially a subset of the BPMN notation [5], a graphical language designed to specify a process in a standardized way. Intuitively, an execution of the process starts at $\bigcirc$ and ends at $\odot$; a *task* is an atomic activity executed by the process; *parallel splits* W_{PS} open parallel parts of the process, whereas *parallel joins* W_{PJ} re-unite parallel branches. *Transitions* are binary relations describing in which order the flow objects (tasks, events and gateways) have to be performed, and determine the *control flow* of the template. For example, in Fig. 3(b) we have a relation of *causality* between tasks t_a and t_b, stating that t_a must take place before t_b happens as t_a achieves some of t_b's preconditions.

An important feature provided by a process template is *concurrency*, i.e., several tasks can occur concurrently. In Fig. 3(c) an example of concurrency between t_1 and t_2 is shown. In order to represent two or more concurrent tasks in a template, the process designer makes use of the parallel gateways, that indicate points of the template in which tasks can be carried out concurrently. A *linearization* of a process template is any linear ordering of the tasks that is consistent with the ordering constraints of the template itself [6]; i.e., a linearization of a partial order is a potential *execution path* of the template from the start event $\bigcirc$ to the end event $\odot$. For example, the template in Fig. 3(c) has two possible execution paths $r_1 = [\bigcirc; t_a; t_1; t_2; t_b; \odot]$ and $r_2 = [\bigcirc; t_a; t_2; t_1; t_b; \odot]$.

Definition 4. *Given a process template* PT *and an initial state* $state_{\mathsf{C}_0} \vdash init_{\mathsf{C}}$*, a state* $state_{\mathsf{C}_i}$ *is said to be* ***reachable*** *with respect to* PT *iff there exists an* *execution path* $r = [\bigcirc; t_1; t_2; ...t_k; \odot]$ *of* PT *and a task* t_i *(with* $i \in 1..k$*) such that* $state_{\mathsf{C}_i} = update(update(\ldots update(state_{\mathsf{C}_0}, Eff_{t_1}) \ldots, Eff_{t_{i-1}}), Eff_{t_i})$.

Definition 5. *A task* t_1 ***affects*** *the execution of a task* t_2 *(*$t_1 \rhd t_2$*) iff there exists a reachable state* $state_{\mathsf{C}_i}$ *of* PT *(for some initial state* $state_{\mathsf{C}_0}$*) such that:*
(i) $state_{\mathsf{C}_i} \vdash Pre_{t_2}$ (ii) $update(state_{\mathsf{C}_i}, Eff_{t_1}) \nvdash Pre_{t_2}$

This means that Eff_{t_1} modify some terms in V that are required as preconditions for making t_2 executable in $state_{\mathsf{C}_i}$.

Definition 6. *Given a process template* PT*, a case* C *and an initial state* $state_{\mathsf{C}_0} \vdash init_{\mathsf{C}}$*, an execution path* $r = [\bigcirc; t_1; t_2; ...t_k; \odot]$ *(where* $k = |T|$*) of* PT *is said to be* ***executable*** *in* C *iff:*
(i) $state_{\mathsf{C}_0} \vdash Pre_{t_1}$ (ii) *for* $1 \leq i \leq k-1$, $update(state_{\mathsf{C}_{i-1}}, Eff_{t_i}) \vdash Pre_{t_{i+1}}$
(iii) $update(state_{\mathsf{C}_{k-1}}, Eff_{t_k}) = state_{\mathsf{C}_k} \vdash goal_{\mathsf{C}}$

Definition 7. *A process template* PT *is said to be* ***executable*** *in a case* C *iff any execution path of* PT *is executable in* C.

Definition 8. *Given a process template* PT*, a task* t_x *is said to be* ***concurrent*** *with a task* t_z *iff there exist two execution paths* r_1 *and* r_2 *of* PT *such that* $r_1 = [\bigcirc; t_1; t_2; ...; t_x; ...; t_z; ...; \odot]$ *and* $r_2 = [\bigcirc; t_1; t_2; ...; t_z; ...; t_x; ...; \odot]$.

Definition 9. *Two concurrent tasks* $t1$ *and* $t2$ *are said to be* ***independent*** *(*$t1 \parallel t2$*) iff* $t1 \not\rhd t2$ *and* $t2 \not\rhd t1$*; that is,* $t1$ *does not affect* $t2$ *and vice versa.*

5 On Synthesizing a Library of Process Templates

Our approach is focussed on the development and use of a *library* of *process templates*. These are reusable processes that achieve specified goals of interest in any starting state that satisfies the template's required preconditions. Specifically, we focus on the use of a POP-based tool that can synthesize complex concurrent process models, while ensuring that concurrent tasks never interfere. The process designer's role is to specify the domain and context in which the template may be executed. Our POP-based tool can then be used to synthesize some candidate process models for the template. If the tool fails to generate a process model or the generated processes are of insufficient quality (e.g., they are too time consuming, unreliable, or lack concurrency), the designer can refine the domain theory and case to obtain better solutions. Once a satisfactory template has been obtained, it is added to the library. The POP-based tool automatically identifies the required preconditions for the template to achieve its goal, meaning the template can be reused whenever a case that matches the template's preconditions arises. The designer maintains the template library over time, in order to have templates that handle effectively most the cases that arise.

The General Framework. Our approach to the definition of a process template (cf. Fig. 4) requires a fundamental shift in how one thinks about

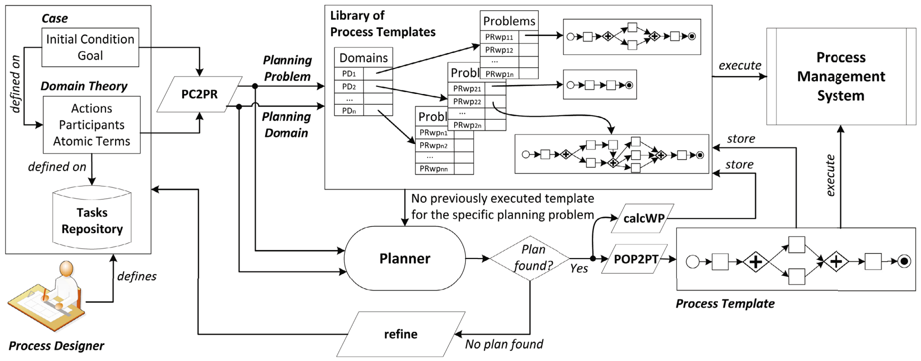


Fig. 4. Overview of the general approach

modeling business processes. Instead of defining a process model "by hand", the process designer has to address her/his efforts to specifying the Domain Theory D and the Case C to be handled. In particular, s/he has to "guess" the starting condition $init_{\mathsf{C}}$, by instantiating only those atomic terms needed for depicting the context s/he has in mind. This means that $init_{\mathsf{C}}$ can be partially specified, i.e, not all terms need to be instantiated with some value. ***Example.*** *Let us consider the scenario depicted in Section 2, represented with a Domain Theory* D_1 *and a goal condition* $goal_{\mathsf{C}_1} = \{fire\_free[loc31]{=}{=}\ true \wedge evacuated[loc32]{=}{=}\ true \wedge\ photo\_taken[loc33]{=}{=}\ true\}$. *Since the process designer may be interested in an emergency process that involves the fewest participants, s/he can start by modeling a starting condition* $init_{\mathsf{C}_1}$ *with information involving only actors act1 and act2 and the robot rb1, while terms involving act3, act4 and rb2 are not explicitly instantiated in* $init_{\mathsf{C}_1}$.

A specific module named PC2PR is in charge of converting the Domain Theory D and the Case C just defined into the corresponding Planning Domain PD and Planning Problem PR specified in PDDL version 2.1[1] (cf. [4]). Basically, PC2PR implements a function $f_{\mathsf{PC2PR}} : (\mathsf{D}, init_{\mathsf{C}}, goal_{\mathsf{C}}) \rightarrow (\mathsf{PD}, init_{\mathsf{PR}}, goal_{\mathsf{PR}})$. Since the use of classical partial-order algorithms for synthesizing the template requires the initial state of PR to be a complete state, we make the closed world assumption [7] and assume that every atomic term $v_j[y_j]$ that is not explicitly specified in $init_{\mathsf{C}}$ is assumed to be false (if $v_j[y_j]$ is a boolean term) or "not assigned" (if $v_j[y_j]$ is a integer or a functional term) in $init_{\mathsf{PR}}$. At the heart of our approach lies a library of process templates built for specific planning domains and problems/cases. If library templates exist for the current values of PD and PR, we can retrieve an appropriate template and allow to execute it through an external PMS. However, if no template exists for the current values of PD and PR, we can invoke an external POP planner on these same inputs. The planner will try to synthesize a plan fulfilling the goal condition $goal_{\mathsf{PR}}$. If the

[1] PDDL 2.1 enables the representation of realistic planning domains, which include operators with universally quantified effects and numeric fluents. However, our formalism does not currently handle conditional effects nor negative preconditions.

planner is unable to find a plan, this suggests there are some missing elements in the definition of the Domain Theory D or in the Case C. Hence, to address this particular case, one can try to *refine* the case C and add information so that it becomes possible to generate a plan. There are many ways to strengthen a problem description, such as adding to the starting condition $init_{\mathsf{C}}$ some terms initially ignored (e.g., to specify the position of every participant), or adding new objects in D or new activities in T (e.g., if a task for extinguish fire is missing). Our approach assumes that one specifies the context step-by-step, and requires the process designer to contribute to the system. ***Example.*** *If the planner is invoked with* $init_{\mathsf{PR}_1}$ *(devised by applying* f_{PC2PR} *on the triple* D_1, $init_{\mathsf{C}_1}$, $goal_{\mathsf{C}_1}$*), it will not be able to find any plan for the specific problem. This is because rb1 does not have enough battery charge for moving, taking pictures and removing debris. The designer can try to add new information to the problem description by instantiating in* $init_{\mathsf{C}_1}$ *all those atomic terms related to actor act4, the only one able to charge robot batteries, and devises a new starting condition* $init_{\mathsf{C}_2}$ *(and, consequently, a new initial planning state* $init_{\mathsf{PR}_2}$*). A planner invoked with* $init_{\mathsf{PR}_2}$ *is finally able to find a consistent plan* P_1 *satisfying* $goal_{\mathsf{PR}_1}$*.*

When the POP planner is able to find a partially ordered plan P consistent with the actual contextual information, three further steps are required. First we need to translate the plan into a template PT that preserves the ordering constraints imposed by the plan. A **solution plan** is a three-tuple $\mathsf{P} = (A, O, CL)$ that specifies the causal relationships for the actions $a_i \in A$, but without specifying an exact order for executing them. Since the actions and the set of ordering constraints must be represented explicitly as nodes and transitions in the template, we developed a module `POP2PT` implementing a function $f_{\mathtt{POP2PT}} : \mathsf{P} \rightarrow \mathsf{PT}$ that takes as input P and converts it into a template PT. It works by first finding the immediate predecessors/successors of actions in the plan using the ordering constraints, and then constructing the desired plan template, inserting parallel splits (resp. join) gateways when an action has more than one immediate successor (resp. predecessor). ***Example.*** *By applying* $f_{\mathtt{POP2PT}}$ *to* P_1*, we devise the template* PT_1 *in Fig. 5(a). Dashed arrows are causal links that imply an ordering constraint between pairs of tasks. For example, the ordering constraint between Go[act1,loc00,loc31] and ExtinguishFire[act1,loc31] is derived from the fact that Go has the effect at[act1]=loc31 that is needed by ExtinguishFire as precondition (i.e., act1 has to be located in loc31 for extinguish the fire in that location).*

Secondly, our approach *infers* the weakest preconditions w_{PT} about the starting state that are required for the template to achieve its goal. The module we use for inferring w_{PT} is called `calcWP` and works by analyzing the set of causal links CL computed by the POP planner, to see which logical facts f_k are involved in causal links that originate from the dummy start action a_0 and end in some $a_k \in A$. More formally:

$$\forall (cl_k, f_k, a_k)\ s.t.\ cl_k = (a_0 \xrightarrow{f_k} a_k) \in CL, then\ f_k \in w_{\mathsf{PT}}. \tag{1}$$

Observe that the effects of $a_0 \in A$ specify all atomic facts that are true in the starting state $init_{\mathsf{PR}}$. The initial facts that are actually required for the plan to be executable and achieve its goal are those that are involved in a causal link with another action in the plan, and we collect those in w_{PT} as specified in Equation 1 (the plan cannot depend on any negative facts as they cannot appear in either the goal or in action preconditions). Basically, w_{PT} is the conjunction of those facts strictly required for executing the plan P (and, consequently, the devised template PT), and is used for devising a new problem $\mathsf{PR}_{wp} = \{w_{\mathsf{PT}}, goal_{\mathsf{PR}}\}$. We can then drop the closed world assumption. For any initial state that satisfies w_{PT}, the obtained process template PT will be executable and achieve the goal condition $goal_{\mathsf{PR}}$. ***Example.*** *If we invoke* `calcWP` *on the causal links devised from* P_1*, we may infer* w_{PT_1}*. Hence, for executing* PT_1 *(cf. Fig. 5(a)) we need to know the positions and capabilities of act1, act2, act4 and rb1; the other contextual information is not strictly needed for a correct execution of the template.*

Thirdly, after the process template PT has been synthesized starting from P, it can be stored in our library together with information about the planning domain PD and abstracted problem PR_{wp}. Specifically, for every different planning domain PD devised through our approach, there is a pointer to a list of different abstracted planning problems PR_{wp} used for obtaining consistent plans in previous executions of our tool, together with the devised process templates. When a process designer defines a new domain theory D_{new} and a case C_{new}, the system checks if the corresponding planning domain PD_{new} and problem PR_{new} (obtained by applying f_{PC2PR} to D_{new} and C_{new}) are already present in our library. If the library contains a planning domain PD and an abstracted planning problem PR_{wp} (together with the associated template PT_{lib}) such that $\mathsf{PD}_{new} = \mathsf{PD}$ and $goal_{\mathsf{PR}} = goal_{\mathsf{PR}_{new}}$ and with $init_{\mathsf{PR}_{new}} \vdash w_{\mathsf{PT}}$, then PT_{lib} is executable respect to PR_{new} (and therefore with respect to C_{new}). This makes our templates reusable in a variety of different situations, in which we don't have complete information about the starting state. At this point, the process designer may decide to execute through an external PMS the template PT_{lib} just found, or to refine D_{new} and C_{new} if PT_{lib} does not fit with the designer expectations. ***Example.*** *Let us suppose that the template shown in Fig. 5(a) does not satisfy at all the process designer, since s/he could add one further robot rb2 to the scenario in order to increase the degree of parallelism in the tasks execution. It follows that a new starting condition* $init_{\mathsf{C}_3}$ *including also contextual information about rb2 can be defined. The associated initial planning state* $init_{\mathsf{PR}_3}$*, together with the original goal condition* $goal_{\mathsf{PR}_1}$ *and the planning domain* PD_1 *are first used for verifying if a previously executed template is already stored in the library. The library returns the template* PT_1 *shown in Fig. 5(a), since its weakest preconditions* w_{PT_1} *are satisfied by* $init_{\mathsf{PR}_3}$ *(i.e.,* $init_{\mathsf{PR}_3} \vdash w_{\mathsf{PT}_1}$*), and goal condition and planning domain are the same as before. Even if the template in Fig. 5(a) is executable with* $init_{\mathsf{PR}_3}$*, the designer may try to search for another plan that (maybe) could exploit the presence of the new robot rb2. The planner builds a new plan starting from* $init_{\mathsf{PR}_3}$*, and the associated template* PT_2 *is shown in Fig. 5(b).* PT_2 *requires the presence of one more robot (i.e., robot rb2) and more contextual information for being executed (so its weakest preconditions* w_{PT_2} *are "richer" than* w_{PT_1}*), but it provides an higher degree of*

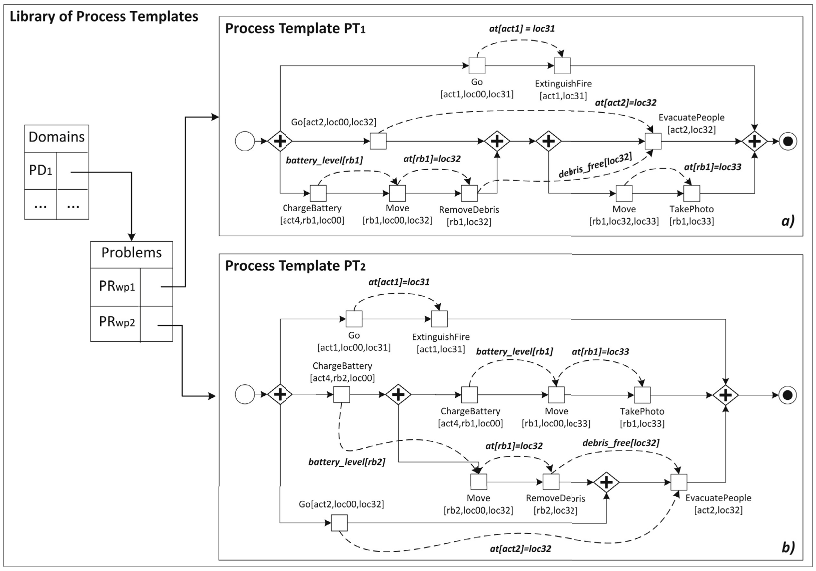


Fig. 5. Templates dealing with the scenario in Fig. 1

concurrency in the execution of its tasks. This means that the process designer can choose which template is the best for her/his purposes: one with less concurrency in the tasks enactment but with the fewest participants (cf., Fig. 5(a)), or one with more concurrency but requiring more resources for being executed (cf., Fig. 5(b)).

Despite the fact that a template is executable "as is", it can be seen as an "intermediate version" of a completely defined process. In fact, the present POP-based tool cannot be used to synthesize templates involving loops or branching on conditions, and the designer may develop these manually by customizing the template to the specifics of the situation.

Properties. A process template PT guarantees some interesting properties, such as the *executability* of the template with respect to the information available in the starting state, and the property of *sound concurrency*, meaning that concurrent activities of a template are proven to be independent from each other.

Theorem 1. *Given a solution plan* P*, a process template* PT *synthesized for* P *using our approach is* ***executable*** *for any process case* C *that satisfies the weakest preconditions* wp_{PT} *inferred from* P*.*

The proof is straightforward. By definition, a sound planner generates a *consistent* plan [3] that leads from an initial state to a goal. Since we represent the Domain Theory/Case as PDDL planning domain/problem, the planner synthesizes a plan (i.e., a process template) that is executable with respect to Definition 7.

A second property we can prove is *sound concurrency.* Even if in a process designed by following data and workflow patterns [8] the concurrent execution of two or more tasks should guarantee the consistency of data accessed by the concurrent tasks, in practice this is often not true. In fact, in complex environments there isn't a clear correlation between a change in the context and corresponding process changes, making it difficult to design by hand a process where concurrent tasks are also independent. On the contrary, all concurrent tasks of a template built with our approach are proven to be *independent* one from another.

Theorem 2. *Given a process template* PT *synthesized with our approach, all concurrent tasks are **independent**.*

Proof. By contradiction, let us suppose that a process template PT has two concurrent tasks t_1 and t_2 such that $t_1 \nparallel t_2$. Hence, t_1 (or t_2) has some effect affecting the precondition of t_2 (or of t_1). This means that $t_1 \triangleright t_2$ or $t_2 \triangleright t_1$. Since PT has been synthesized as result of a POP planner, this dependency between t_1 and t_2 would be represented with a causal link $t_1 \xrightarrow{e} t_2$ (or $t_2 \xrightarrow{e} t_1$), where e is an effect of task t_1 and a precondition for task t_2 (or vice-versa). This causal link requires an ordering between t_1 and t_2, meaning they need to be executed (and represented in the process template) in sequence. But this means that t_1 and t_2 are not concurrent tasks, by contradicting the original hypothesis. □

Experiments. To show the feasibility of our approach, we ran some experiments and measured the time required for synthesizing a partially ordered plan for some variants of our running example described in Section 2. We ran our tests using POPF2 [9] on an Intel U7300 1.30GHz, 4GB RAM machine. POPF2 is a temporal planner that handles PDDL 2.1 [4] and preserves the benefits of partial-order plan construction in terms of producing makespan-efficient, flexible plans.

The experimental setup was run on variants of our running example. We represented 7 planning actions in PD (corresponding to 7 different tasks stored in the tasks repository T), annotated with 7 relational predicates and 6 numeric fluents, in order to make the planner search space sufficiently challenging. Then, we defined 18 different planning problems of varying complexity by manipulating the number of facts in the goal. As well, we examined how irrelevant domain knowledge affects the performance of the planner. Starting from a planning problem PR with an initial state $init_{\mathsf{PR}}$ completely specified and with a goal condition $goal_{\mathsf{PR}}$ expressed as the conjunction of n facts, we manipulated the specification of the initial state $init_{\mathsf{PR}}$ to reduce the number of known facts. In our experiments, the number of facts in goal condition ranges from 1 single fact to a conjunction of 6 logical facts (that make the contextual problem harder). As shown in Table 1, for a given goal condition composed of n facts, our purpose was to measure the computation time needed for finding a sub-optimal solution for problems specified with starting states with a decreasing amount of knowledge. The column labeled as "Knowledge in $init_{\mathsf{PR}}$" makes explicit which information is removed from the initial state of the planning problem. For example, if we consider our running scenario from Section 2, whose goal condition is composed of 3 facts and characterized by a complete specification of the starting state, the time needed for finding a solution plan is of 0.13 seconds. After removing from the initial state all the information concerning the actor *act3*, the time

Table 1. Time performances of POPF2

Facts in $goal_{\mathsf{PR}}$	Knowledge in $init_{\mathsf{PR}}$	Time for a sub-opt. sol.
	complete state	0.17
1	No information about act1	0.15
	No information about act1 and act3	0.12
	complete state	0.12
2	No information about act3	0.10
	No information about act3 and rb1	0.08
	complete state	0.13
3	No information about act3	0.11
	No information about act3 and rb2	0.09
	complete state	0.21
4	No information about act3	0.20
	No information about act3 and rb1	0.10
	complete state	0.17
5	No information about act3	0.16
	No information about act3 and rb1	0.10
	complete state	1.56
6	No information about act3	1.19
	No information about act3 and act1	1.13

required for computing the plan decreases to 0.11 seconds. In general, for a given goal condition, removing "irrelevant information" from the initial state reduces the search space and the computation required for synthesizing the plan. Note that a sub-optimal solution includes more actions than those strictly required for fulfilling the goal, and when the number of facts in a goal condition increases, the quality of the solution may decrease. Based on our experiments, the approach seems feasible for medium-sized dynamic processes as used in practice.

6 Related Work

Process modeling is the first and most important step in the BPM lifecycle [1], which intends to provide a high-level specification of a business process that is independent from implementation and serves as a basis for process automation and verification. The task of defining a model is often performed with the aid of tools that provide a graphical representation, but without any automatic generation of the process model. However, in recent years, numerous AI planning-based approaches have been devised for the latter, and the closest to our approach are [10,11,12]. [10] presents the basic idea behind the use of planning techniques for generating a process schema, but no implementation seems to be provided, and the direct use of the PDDL language for specifying the domain theory requires a deep understanding of AI planning technology. In [11], the authors exploit the IPSS planner for modeling processes in SHAMASH [13], a knowledge-based system that uses a rule-based approach. To automate the process model generation, they first translate the semantic representation of SHAMASH into the IPSS language. Then, IPSS produces a parallel plan of activities that is finally translated back into SHAMASH and is presented graphically to the user. However, the emphasis is on supporting processes for which one has complete knowledge, while for dynamic processes some contextual information may not be available at the time of process model synthesis. The work of [12] is based on learning activities as planning operators and feeding them to a planner that generates the process model. An interesting result concerns the possibility of producing process models even though the activities may not be accurately described. In such cases, the authors use a best-effort planner that is always able to create a plan, even though the plan may be incorrect. After a finite number of refinements, the best candidate plan (i.e., the one with the lowest number of unsatisfied preconditions) is translated into a process model. Unfortunately, the best plan found is often far from the correct solution [12].

7 Conclusion

In this paper, we developed a technique based on POP algorithms and declarative specifications of process tasks for synthesizing a library of process templates to be enacted in partially specified contextual scenarios. We are currently working on a complete implementation and thorough validation of the approach, including the formalization of metrics for evaluating process templates' quality. A future direction for this work is to generate hierarchical process templates, with high-level templates achieving more general goals that can invoke simpler templates to achieve some of their subgoals. We also plan to address expressiveness limitations, such as handling preferences and representing negative preconditions.

References

1. Weske, M.: Business Process Management: Concepts, Languages, Architectures. 2nd edn. Springer (2010)
2. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems. Springer, Berlin (2012)
3. Weld, D.: An Introduction to Least Commitment Planning. AI Mag. 15(4) (1994)
4. Fox, M., Long, D.: PDDL2.1: an Extension to PDDL for Expressing Temporal Planning Domains. J. Artif. Int. Res. 20(1) (2003)
5. White, S.A., Miers, D.: BPMN Modeling and Reference Guide: Understanding and Using BPMN. Future Strategies Inc. (2008)
6. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer (1996)
7. Reiter, R.: On Closed World Data Bases. In: Ginsberg, M. (ed.) Readings in Nonmonotonic Reasoning, Morgan Kaufmann Publishers Inc. (1987)
8. Dumas, M., van der Aalst, W.M.: Process-aware information systems: bridging people and software through process technology. Wiley-Interscience (2005)
9. Coles, A.J., Coles, A., Fox, M., Long, D.: Forward-Chaining Partial-Order Planning. In: ICAPS (2010)
10. Schuschel, H., Weske, M.: Triggering replanning in an integrated workflow planning and enactment system. In: Benczúr, A.A., Demetrovics, J., Gottlob, G. (eds.) ADBIS 2004. LNCS, vol. 3255, pp. 322–335. Springer, Heidelberg (2004)
11. R-Moreno, M.D., Borrajo, D., Cesta, A., Oddi, A.: Integrating Planning and Scheduling in Workflow Domains. Exp. Syst. with App.: An Int. J. 33(2) (2007)
12. Ferreira, H., Ferreira, D.: An Integrated Life Cycle for Workflow Management Based on Learning and Planning. Int. J. Coop. Inf. Systems 15 (2006)
13. Aler, R., Borrajo, D., Camacho, D.: A Knowledge-based Approach for Business Process Reengineering, SHAMASH. Know.-Based Syst. 15(8) (2002)