Continuous Planning for Solving Business Process Adaptivity

Andrea Marrella and Massimo Mecella

Dipartimento di Informatica e Sistemistica Antonio Ruberti,
SAPIENZA - Universitá di Roma, Italy
{marrella,mecella}@dis.uniroma1.it

Abstract. Process Management Systems (PMSs, aka Workflow Management Systems – WfMSs) are currently more and more used as a supporting tool to coordinate the enactment of processes. In real world scenarios, the environment may change in unexpected ways so as to prevent a process from being successfully carried out. In order to cope with these anomalous situations, a PMS should automatically adapt the process without completely replacing it. In this paper, we propose a technique, based on continuous planning, to automatically cope with unexpected changes, in order to modify only those parts of the process that need to be changed/adapted and keeping other parts stable. We also provide a running example that shows the practical applicability of the approach.

Key words: Processes, Adaptivity, Continuous Planning, Process Management Systems

1 Introduction

Process Management Systems (PMSs, aka Workflow Management Systems) [1] are applied to support and automate process enactment, aiming at increasing the efficiency and effectiveness in its execution. Classical PMSs offer good process support as long as the processes are structured and do not require much flexibility. In the last years, the trade-off between *flexibility* and *support* has become an important issue in workflow technology [2]. If on the one hand there is a desire to control processes and avoid incorrect executions of the processes, on the other hand users want flexible processes that do not constraint them in their action. A recent open research question concerns how to tackle scenarios characterized by being very dynamic and subject to higher frequency of unexpected contingencies than classical scenarios, e.g., a scenario for emergency management. There, a PMS can be used to coordinate the activities of emergency operators within teams. Figure 1 shows a (slightly simplified) example of a possible scenario for the aftermath of an earthquake. The process depicts some actors that assess an area for dangerous partially-collapsed buildings. Meanwhile others are giving first aid to the injured people and filling in a questionnaire. In such a context, the PMS must provide an high degree of both support and flexibility. Hence, if on the one hand it should "drive" each actor along the control flow, by guaran-

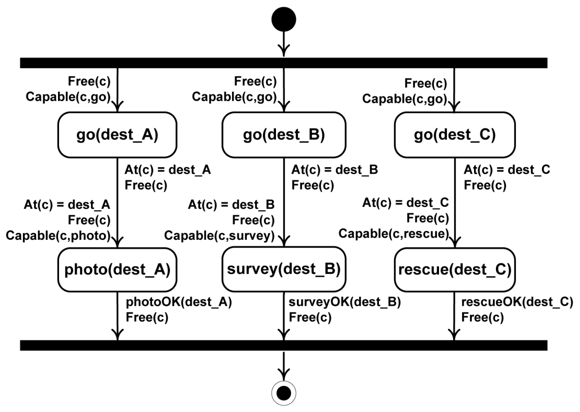


Fig. 1. A possible process to be carried on in disaster management scenarios

teeing that each task in the process is executed in the correct order and with a proper termination, on the other side it should automatically react to exceptions by adapting the process without completely replacing it. This paper proposes a novel approach, in which *continuous planning* [3] is used to improve the degree of *automatic* adaptation in PMSs. The technique, which constitutes an interesting application of non-classical planning, is able to automatically adapt processes without explicitly defining handlers/policies to recover from exogenous events. In order to describe our approach, we make use of a declarative model named SmartPM. Our model allows to define logical constraints and provides a proper execution engine that manages the process routing and decides which tasks are enabled for execution, by taking into account the control flow and the value of variables. Once a task is ready for being assigned, the engine is also in charge of assigning it to a proper service (which may be a human actor, a robot, a software application, etc.). In contrast with other approaches that use planning to handle adaptivity [4, 5], our technique provides two interesting features in recovering failures : *(i)* it modifies only those parts of the process that need to be adapted by keeping other parts stable; *(ii)* it is a non-blocking technique; it does not stop directly any task in the main process during the computation of the recovery process. The rest of the paper is organized as follows. Section 2 covers the state of the art in adaptivity in PMSs and relevant results in planning. Sections 3 and 4 introduce the basic preliminary concepts, whereas Sections 5 and 6 illustrate both the general framework and the technique for automatically adapting processes. Finally, Section 7 concludes the paper.

2 Related Works

The approach proposed in this paper tackles the problem of automatic adaptivity in PMSs by using techniques devised from the planning community. Adaptivity in PMSs concerns the capability to face *exceptional changes*, which are char-

Table 1. Features provided by the leading PMSs to manage adaptation

Product	**Manual**	**Pre-planned**	**Unplanned**
YAWL [6]		✓	
DECLARE [2]		✓	
OPERA [7]	✓	✓	
ADEPT2 [8]	✓		
ADOME [9]	✓		
AgentWork [10]	✓		
ProCycle [11]	✓		
WASA [12]	✓		
SmartPM + Continuous Planning			✓

acterized by events, foreseeable or unforeseeable, during the process instance executions which may require instances to be adapted in order to be carried out. There are two ways to handling exceptional events: *manual* (once events are detected, a responsible person, expert on the process domain, modifies manually the affected instances) or *automatic* (when exceptional events are sensed, PMS is able to change accordingly the schema of affected instances in a way they can still be completed). In the range of automatic adaptation, we can distinguish between two further categories: *pre-planned* adaptation (i.e., for each kind of failure that is envisioned to occur, a specific contingency process is defined a priori) and *unplanned* adaptation. In the latter case, process schemas are defined as if failures could never occur; there is a monitor which is continuously looking for the occurrence of failures. When some of them occur, the process is automatically adapted to mitigate the effects. The difference with the pre-planned adaptation consists in that there exist no pre-planned policies, but the policy is built on the fly for the specific occurrence. Over the years, a multitude of adaptive PMSs (either commercial or research proposals/prototypes) have been developed. Table 1 compares the degree of adaptability to exceptional changes that is currently provided by the leading PMSs (either commercial or research proposals/prototypes). Among them, interesting approaches are ProCycle [11] and ADEPT2 [8]. The first uses a case-based reasoning approach to support adaptation of workflow specifications to changing circumstances. Case-based reasoning (CBR) is the way of solving new problems based on the solutions of similar past problems: users are supported to adapt processes by taking into account how previously similar events have been managed. However, adaptation remains manual, since users need to decide how to manage the events though they are provided with suggestions. ADEPT2 features a check of "semantic" correctness to evaluate whether events can prevent processes from completing successfully. But the semantic correctness relies on some semantic constraints that are defined manually by designers at design-time and are not inferred, e.g., over pre- and post-conditions of tasks. Pre-planned approaches to exceptional changes (a.k.a. exceptions) are often based on the specification of exception handlers and compensation flows [7], with the challenge that in many cases the compensation cannot be performed by simply undoing actions and doing them again. Our approach is complementary with regard to this literature, and leverages on it for dealing with exceptional changes that can be pre-planned. The novelty is that we propose, in addition to incorporating the previous techniques in a PMS, also to consider automatic adaptation to unplanned exceptions.

2.1 Planning Algorithms

Planning systems are problem-solving algorithms that operate on explicit representations of states and actions. The standard representation language of classical planners is known as the Planning Domain Definition Language [13] (PDDL); it allows to formulate a problem through the description of the initial state of the world, the description of the desired goal and a set of possible actions. An *action definition* defines the condition under which an action can be executed, called *pre-conditions* and its effects on the state of the world, called *effects*. The set of all action definitions represents the *domain* of the planning problem. A planner that works on such inputs generates a sequence of actions (the *plan*) that leads from the initial state to a state fulfilling the goal. The code in Figure 2b depicts the PDDL representation of the task $go(i)$ (it is the first task defined in each branch of the process in Figure 1) that instructs a service c to move towards the destination denoted by i. *Continuous Planning* [3] refers to the process of planning in a world under continual change, where the planning problem is often a matter of adapting to the world when new information is sensed. A continuous planner is designed to persist indefinitely in the environment. Rather than thinking of the planner and execution monitor as separate processes, one of which passes its results to the other, we can think of them as a single process. In order to validate our approach, we make use of a continuous planner working on top of UCPOP planner [14].

3 Preliminaries

In this paper, we use the situation calculus (SitCalc) to formalize adaptation in PMSs. The SitCalc is a second-order logic formalism designed for representing and reasoning about dynamic domains [15]. In the SitCalc, a dynamic world is modeled as progressing through a series of *situations* as a result of various *actions* being performed. A situation represents a history of actions occurred so far. The constant S_0 denotes the initial situation, and a special binary function symbol $do(a, s)$ denotes the next situation resulting from the performance of action a in situation s. Conditions whose truth value may change are modeled by means of *fluents*. Technically, these are predicates taking a situation term as their last argument. Fluents may be thought of as “properties” of the world whose values may vary across situations. Changes in fluents (resulting from executing actions) are specified through *successor state axioms*. In particular for each fluent F we have a successor state axioms as follows:

$$F(\overrightarrow{x}, do(a, s)) \Leftrightarrow \Gamma_F(\overrightarrow{x}, a, s) \tag{1}$$

Table 2. IndiGolog constructs

Construct	**Meaning**
a	A primitive action.
$\sigma?$	Wait while the σ condition is false.
$(\delta_1; \delta_2)$	Sequence of two sub-programs δ_1 and δ_2.
$proc\ P(\overrightarrow{v})\ \delta$	Invocation of a IndiGolog procedure δ passing a vector $\overrightarrow{v}$ of parameters.
$(\delta_1 \vert \delta_2)$	Non-deterministic choice among (sub-)program δ_1 and δ_2.
$if\ \sigma\ then\ \delta_1\ else\ \delta_2$	Conditional statement: if σ holds, δ_1 is executed; otherwise δ_2.
$while\ \sigma\ do\ \delta$	Iterative invocation of δ.
$(\delta_1 \parallel \delta_2)$	Concurrent execution.
δ^*	Non-deterministic iteration of program execution.
$\pi a.\delta$	Non-deterministic choice of argument a followed by the execution of δ.
$\langle \sigma \rightarrow \delta \rangle$	δ is repeatedly executed until σ becomes false, releasing control to anyone else able to execute.
$send(\Upsilon, \overrightarrow{v})$	a vector $\overrightarrow{v}$ of parameters is passed to an external program Υ.
$receive(\Upsilon, \overrightarrow{v})$	a vector $\overrightarrow{v}$ of parameters is received by an external program Υ.

where $\Gamma_F(\overrightarrow{x}, a, s)$ is a formula with free variables fully capturing the truth-value of fluent F on objects $\overrightarrow{x}$ when action a is performed in situation s. Besides successor state axioms, SitCalc is characterized by *action precondition axioms*, which specify whether a certain action is executable in a situation. Action precondition axioms have the form:

$$Poss(a, s) \Leftrightarrow \Pi_a(s) \tag{2}$$

where the formula $\Pi_a(s)$ defines the conditions under which the action a may be performed in the situation s. In order to control the execution of actions we make use of high level programs, expressed in Golog-like programming languages. In particular we focus on IndiGolog [16], a programming language for autonomous agents that sense their environment and act as they operate. The programmer provides a high-level nondeterministic program involving domain-specific actions and tests to perform the tasks. The IndiGolog interpreter reasons about the preconditions and effects of the actions in the program to find a legal terminating execution. To support this, the programmer provides a SitCalc *theory*, that is a declarative specification of the domain (i.e., primitive actions, preconditions and effects, what is known about the initial state) in the situation calculus. IndiGolog is equipped with standard imperative constructs (e.g., sequence, conditional, iteration, etc.) as well as procedures and primitives for expressing various types of concurrency and prioritized interrupts. The Table 2 summarizes the constructs of IndiGolog used in this work. Basically, these constructs allow to define every well-structured process as defined in [17]. Let's focus on the interrupt construct:

$$\begin{array}{l}\langle\ \sigma \rightarrow \delta\ \rangle \overset{\text{def}}{=} \textbf{while}\ Interrupts\_running\ \textbf{do} \\ \textbf{if}\ \sigma\ \textbf{then}\ \delta\ \textbf{else}\ false\ \textbf{endIf} \\ \textbf{endWhile}\end{array}$$

To see how this works, first assume that the special fluent $Interrupts\_running$ is identically true. When an interrupt $\langle \sigma \rightarrow \delta \rangle$ gets control from higher priority processes, suspending any lower priority processes that may have been advancing, it repeatedly executes δ until σ becomes false. Once the interrupt body

δ completes its execution, the suspended lower priority processes may resume. The control release also occurs if σ cannot progress (e.g., since no action meets its precondition). The interrupt construct allows IndiGolog to provide a formal notion of *interleaved planning, sensing, and action.* Roughly speaking, an *online execution* of a program finds a next possible action, executes it in the real world, obtains sensing information afterward, and repeats the cycle until the program is finished. The fact that actions are quickly executed without much deliberation and sensing information is gathered after each step makes the approach realistic for dynamic and changing environments. Finally, IndiGolog provides flexible mechanisms for interfacing with other programming languages such as `Java` or `C`, and for socket communication. For our convenience, we have defined here two more abstract constructs to send/receive parameters as well as values with external programs, defined out of the range of the IndiGolog process. For more details about the communication between IndiGolog and external programs, we refer the reader to [16].

4 Process Formalization in Situation Calculus

When using IndiGolog for process management, we take tasks to be predefined sequences of actions and processes to be IndiGolog programs. The monitoring of the process execution is in charge of SmartPM, our declarative PMS deployed by using the IndiGolog interpreter. It drives the task assignment to services involved in the process execution and repairs the process if it is invalidated. To denote the various objects of interest, we make use of the following domain-independent predicates (that is, non-fluent rigid predicates):

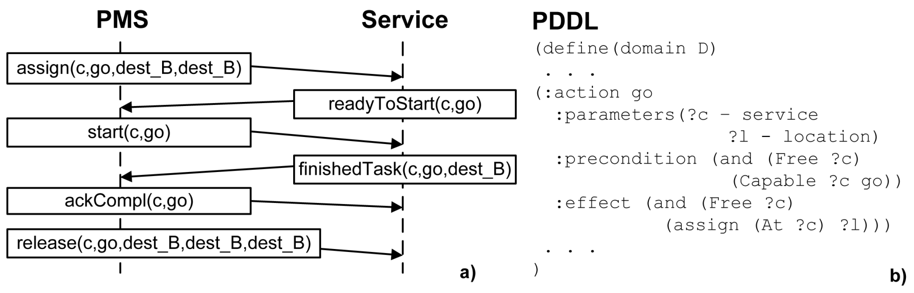
– $Service(c)$: c is a service;
– $Task(t)$: t is a task;
– $Capability(b)$: b is a capability;
– $Provides(c, b)$: service c provides the capability b;
– $Requires(t, b)$: task t requires the capability b.

To refer to the ability of a service c to perform a certain task t, we introduce the following abbreviation:

$$Capable(c, t) \stackrel{\text{def}}{=} \forall b.Requires(t, b) \Rightarrow Provides(c, b). \tag{3}$$

That is, service c can carry out a certain task t iff c provides all capabilities required by the task t. The life-cycle of a task involves the execution of four basic actions:

– $assign(c, t, i, p)$: a task t with input i is assigned to a service c. p denotes the *expected output* that t is supposed to return if its execution is successful;
– $start(c, t)$: service c is notified to start task t;
– $ackCompl(c, t)$: service c acknowledges of the completion of task t;
– $release(c, t, i, p, q)$: service c releases task t, executed with input i and expected output p, and returns an output q.


PMS
Service
assign(c,go,dest_B,dest_B)
readyToStart(c,go)
start(c,go)
finishedTask(c,go,dest_B)
ackCompl(c,go)
release(c,go,dest_B,dest_B,dest_B)
a)
PDDL
b)

```
(define(domain D)
 . . .
(:action go
  :parameters(?c - service
                   ?l - location)
  :precondition (and (Free ?c)
                   (Capable ?c go))
  :effect (and (Free ?c)
             (assign (At ?c) ?l)))
 . . .
)
```

Fig. 2. a) The life cycle and b) the PDDL representation of the task *go*

The terms i, p and q denote arbitrary sets of input/output, which depend on the specific task. The actions performed by the process need to be "complemented" by other actions executed by the services themselves. The following are used to inform the PMS engine about how tasks execution is progressing:

– $\mathit{readyToStart}(c, t)$: service c declares to be ready to start performing task t;
– $\mathit{finishedTask}(c, t, q)$: service c declares to have completed the execution of task t with output q.

Figure 2a depicts the protocol for a successful execution of a generic task $go(i)$, that instructs a service c to move towards the destination denoted by $i = dest_B$. Note that we suppose to work with domains in which services, tasks, input and output parameters are finite. The formalization of processes in IndiGolog requires two distinct sets of fluents. A first set includes those "engine fluents" that the PMS uses to manage the task life-cycle of processes (for the sake of brevity, here we omit their specification). The second set concerns such fluents used to denote the data needed by process instances; their definitions depends strictly on the specific process domain of interest. These "data fluents" can be used to constrain the task assignment, to record the outcome of a task and as guards into the expressions at decision points (e.g., for cycles, conditional statements). So, if X is a process variable meant to capture the outcome of a (specific) task T, then a SitCalc theory shall include a *data fluent* X_φ[1] with the following successor state axiom:

$$\begin{array}{l} X_\varphi(i, do(a,s)) = q \equiv \\ \quad \big(\exists c, p.\ a = \mathit{release}(c, T, i, p, q)\big) \vee \\ \quad \big(X_\varphi(i,s) = q\ \wedge \neg\exists c, p, q'.\ a = \mathit{release}(c, T, i, p, q') \wedge (q' \neq q)\big). \end{array} \tag{4}$$

The value of X_φ is changed to value q when one of the corresponding tasks finishes with output q. The formalization allows also to define tasks whose associated data fluents can be customized in according to the process needs. For example, one can require some fluents defined for each service c in the formalization. This is the case of the task $go(i)$, just introduced above. The fluent At_φ

[1] Sometimes we use *arguments-suppressed* formulas, i.e., formulas with all arguments suppressed (e.g. X_φ denotes the arguments-suppressed expression for $X_\varphi(i, s)$).

is meant to capture the new position q of the service c in the situation s after the execution of $go(i)$.

$$\begin{aligned}&At_{\varphi}(c, do(a, s)) = q \equiv \\ &\big(\exists i, p.\ a = release(c, go, i, p, q)\big) \vee \\ &\big(At_{\varphi}(c, s) = q \ \wedge \neg\exists i, p, q'.\ a = release(c, go, i, p, q') \wedge (q' \neq q)\big).\end{aligned} \tag{5}$$

As far as it concerns the task assignment, it is driven by the special fluent $Free(c, s)$, which states if a service c is available in situation s for task assignment:

$$\begin{aligned}Poss(assign(c, t, i, p), s) \Leftrightarrow Capable(c, t) \wedge \\ Free(c, s) \wedge (X_{\varphi,1} \wedge ... \wedge X_{\varphi,m}).\end{aligned} \tag{6}$$

A task t can be assigned to a service c iff c is *Free* and is *Capable* to execute t. Moreover, values of data fluents $X_{\varphi,j}$ (where j ranges over {1..m}) possibly included in the axiom should be evaluated.

Example 1. Consider the process instance depicted in Figure 1. In order to represent pre- and post-conditions of each task defined in the control-flow, the following data fluents are needed : *(i)* $At_{\varphi}(c, s)$ stores the location in which the service c is located in situation s; *(ii)* $SurveyOK_{\varphi}(d, s)$ is true in situation s if a survey concerning injured people at destination d has been successfully filled and forwarded to the headquarter; *(iii)* $PhotoOK_{\varphi}(d, s)$ is true if the pictures taken in location d are judged as having a good quality; *(iv)* $RescueOK_{\varphi}(d, s)$ is true if injured people in area d have been supported through a medical assistance. ■

5 General Framework

Before a process starts its execution, the PMS takes the initial context from the real environment and builds the knowledge base corresponding to the initial situation S_0. In S_0 every service is assumed as free. As far as data fluents, their initial value should be defined manually at design-time, since they depend on the specific domain. The PMS also builds an IndiGolog program δ_0 corresponding to the process to be carried on. For simplicity, we consider for the discussion only *well-formed processes* as defined in [17]. Process adaptivity can be seen as the ability of PMS to reduce the gap from the *expected reality* – the (idealized) model of reality that is used by the PMS to deliberate – and the *physical reality*, the real world with the actual values of conditions and outcomes. Roughly speaking, the PMS should be able to find a recovery process δ_h that repairs δ_0 and remove the gap between the two kinds of reality. A first approach in this direction was developed in [18]. That approach synthesizes a linear process δ_h (i.e., a process constituted only by a sequence of actions) inserted at a given point of the original process – exactly the point in which the deviation is identified. In more details, let's assume that the current process is $\delta_0 = (\delta_1; \delta_2)$ in which δ_1 is the part of the process already executed and δ_2 is the part of the process which remain to be executed when a deviation is identified. Then the technique devised in [18] synthesizes a linear process δ_h that deals with the deviation; the adapted process is $\delta_0' = (\delta_1; \delta_h; \delta_2)$. However, whenever a process needs to be adapted, every running task is interrupted, since the "repair" sequence of actions $\delta_h = [a_1, \ldots, a_n]$ is placed before them. Thus, all the branches can only resume execution after the repair sequence has been executed. A slight improvement to the last approach was devised in [19], where the technique is refined by "assuming" concurrent branches as independent (i.e., neither working on the same variables nor affecting some conditions). If independent, it allows to automatically synthesize a linear recovery process such that it affects only the branch interested in the deviation. Hence, if the current process is $\delta_0 = (\delta_1 || \delta_2)$ and the branch δ_2 breaks, the approach proposed in [19] synthesizes a recovery process δ_h such that the adapted process is $\delta_0' = (\delta_1 || (\delta_h; \delta_2))$. Note that also this last approach needs to block the execution of the main process δ_0 until the building of the recovery process δ_h is completed.

The technique proposed in this paper tries to overcome the above limitations by introducing a *non-blocking* repairing technique. The idea is to build the recovery procedure δ_h *in parallel* with the execution of the main process δ_0, avoiding to stop directly any task in the process. Once ready, δ_h will be inserted as a new branch of δ_0 and will be executed in concurrency with every other task. Let's now detail how the proposed technique works. We start by formalizing the concepts of *physical reality* and *expected reality.*

Definition 1 *A physical reality $\Phi(s)$ is the set of all data fluents $X_{\varphi,j}$ (where j ranges over {1..m}) defined in the SitCalc theory. Hence, $\Phi(s) = \bigcup_{j=1..m}\{X_{\varphi,j}\}$.*

The physical reality $\Phi(s)$ captures the values assumed by each *data fluent* in the situation s. Such values reflect what is happening in the real environment whilst the process is under execution. However, the PMS must guarantee that each task in the process is executed correctly, i.e., with an output that satisfies the process specification. For this purpose, the concept of *expected reality* $\Psi(s)$ is needed. For each fluent X_φ, we introduce a new *expected fluent* X_ψ that is meant to record the "expected" value of X after the execution of a task T. The successor state axiom for this new fluent is straightforward:

$$\begin{array}{l} X_\psi(i, do(a, s)) = p \equiv \\ \quad \big(\exists c, q.\ a = release(c, T, i, p, q)\big) \lor \\ \quad \big(X_\psi(i, s) = p\ \land \neg\exists c, p', q.\ a = release(c, T, i, p', q) \land (p' \neq p)\big). \end{array} \qquad (7)$$

It states that in the expected reality a task is *always executed correctly* and forces the value of X_ψ to the value of the expected output p.

Definition 2 *An expected reality $\Psi(s)$ is the set of all expected fluents $X_{\psi,j}$ (where j ranges over {1..m}) defined in the SitCalc theory. Hence, $\Psi(s) = \bigcup_{j=1..m}\{X_{\psi,j}\}$.*

A recovery procedure is needed if the two realities are different from each other, i.e., some tasks in the process failed their execution by returning an output q whose value is different from the expected output p. Since the PMS has to guarantee that each task is executed correctly, if a discrepancy occurs it derives a flow of repairing actions that turns the physical reality into the expected reality. Formally, a situation s is known as Relevant - candidate for adaptation - iff :

$$\mathsf{Relevant}(\delta_0, s) \equiv \neg \mathsf{SameState}(\Phi(s), \Psi(s)) \tag{8}$$

Predicate $\mathsf{SameState}(\Phi(s), \Psi(s))$ holds iff the states[2] denoted by $\Phi(s)$ and $\Psi(s)$ are the same. Each task defined in δ_0 affects (or is affected by) only a finite number of fluents. This means that each task is interested only in that fragment of reality it contributes to modify.

Definition 3 *A task T affects a data/expected fluent X iff $\exists c, i, p, q, a$ s.t. $a = release(c, T, i, p, q)$ and $X(i, do(a, s)) \Leftrightarrow \Gamma_X(i, a, s)$ (cf. equation 1). We denote it with $T \triangleright X$.*

Definition 4 *A task T is affected by a data/expected fluent X iff $\exists c, i, p, a$ s.t. $a = assign(c, T, i, p)$ and $X \in \Pi_a(s)$ (cf. equation 2). We denote it with $T \triangleleft X$.*

The two latter definitions allow to state a new further definition of $\Phi(s)$ and $\Psi(s)$, whose range can be limited to a specific task T.

Definition 5 *Given a specific task T, a T-limited physical reality $\Phi|_T(s)$ is the set of those data fluents $X_{\varphi,j}$ (where j ranges over $\{1..m\}$) such that $T \triangleright X_{\varphi,j}$ or $T \triangleleft X_{\varphi,j}$. We denote these fluents as $X_{\varphi|_T}$. Hence, $\Phi|_T(s) = \bigcup_{j=1..m}\{X_{\varphi,j|T}\}$ and $\Phi|_T(s) \subseteq \Phi(s)$.*

Definition 6 *Given a specific task T, a T-limited expected reality $\Psi|_T(s)$ is the set of those expected fluents $X_{\psi,j}$ (where j ranges over $\{1..m\}$) such that $T \triangleright X_{\psi,j}$ or $T \triangleleft X_{\psi,j}$. We denote these fluents as $X_{\psi|_T}$. Hence, $\Psi|_T(s) = \bigcup_{j=1..m}\{X_{\psi,j|T}\}$ and $\Psi|_T(s) \subseteq \Psi(s)$.*

From definitions 5 and 6, the following one stems :

Definition 7 *Let $T_1, ..., T_n$ all tasks defined in the SitCalc theory. A physical (expected) reality $\Phi(s)$ $(\Psi(s))$ is the union of all T-limited physical (expected) realities that hold in situation s : $\Phi(s) = \bigcup_{i=1..n} \Phi|_{T_i}(s)$ $(\Psi(s) = \bigcup_{i=1..n} \Psi|_{T_i}(s))$.*

Now, the predicate Relevant can be refined in a way that focuses on a specific task T:

$$\mathsf{Relevant}_T(\delta_0, s) \equiv \neg \mathsf{SameState}(\Phi|_T(s), \Psi|_T(s)) \tag{9}$$

Our framework is able to capture - and to recover from - two different kinds of task failure. An *internal failure* is related to the failure in the execution of a task, i.e., the task does not terminate, or it is completed with an output that differs from the expected one. Example 2 shows such a case. An *external failure* is represented as an exogenous event e, given in input by the external

[2] Given a situation s and a set $\overrightarrow{F}$ of fluents, a $state(\overrightarrow{F}(s))$ is the set composed by the values - in s - of each fluent F_j that belongs to $\overrightarrow{F}$. Hence, $state(\overrightarrow{F}(s)) = \bigcup_{j=1..m}\{F_j\}$ *s.t.* $F_j \in \overrightarrow{F}$.

environment, that forces a set of data fluents to assume a value imposed by the event itself. Such a new value could differ from the expected one, by generating a discrepancy between the two realities. In order to capture the effects of e, the process designer has to refine the successor state axiom of those fluents whose value can be affected by e. An example of how to catch an exogenous event is shown in the following section.

Example 2. Consider the process instance depicted in Figure 1. Let's suppose that the PMS assigns the task $go(dest_B)$ to the service $srvc$. Assume that $srvc$, instead to reach $dest_B$, ends the execution of the task *go* in $dest_Z$. Then, after the *release* action is executed, the fluent At_φ takes the value $dest_Z$. But this output does not satisfy the expected outcome. The expected output $p = dest_B$ is stored in the fluent At_ψ; it generates a discrepancy between $\Phi|_{go}(s)$ and $\Psi|_{go}(s)$. This means that the $\mathsf{Relevant}_{go}(\delta_0, s)$ holds, and the main process δ_0 needs to be adapted. ■

6 The Repairing Technique

We now turn our attention to how adaptation is meant to work in our approach. Before starting the execution of the process δ_0, the PMS builds the PDDL representation of each task defined in the SitCalc theory and sends it to an external planner that implements the POP algorithm. In Figure 3, we show how the PMS has been concretely coded by the interpreter of IndiGolog. This framework can be viewed as a dynamic system in which the PMS continually generates new goals in response to its perceptions about physical reality. The main procedure involves three concurrent programs in priority. At a lower priority, the system runs the actual IndiGolog program representing the process to be executed, namely procedure **Process**. This procedure relies, in turn, on procedure **ManageExecution**, which includes task assignment, start signaling, acknowledgment of completion, and final release. The monitor, which runs at higher priority, is in charge of monitoring changes in the environment and adapting accordingly. The first step in procedure **Monitor** checks whether fluent *RealityChanged* holds true, meaning that a service has terminated the execution of a task or an exogenous (unexpected) action has occurred in the system. Basically, the procedure **Monitor** is enabled when the physical or the expected reality (or both) change. If it happens, the monitor calls the procedure **IndiPOP**, whose purpose is to manage the execution of the external planner by updating its initial states and expected goals according with changes in the two realities. **IndiPOP** first builds the two sets *Start* (the initial state) and *Finish* (the goal), by making them equal respectively to $\Phi(s)$ and $\Psi(s)$. As far as concerns the initial state, it will include, for each task t and service c defined in SitCalc theory, the values of $Capable(c, t)$ and of $Free(c, s)$ in addition to the values of data fluents. Then **IndiPOP** catches the partial plan $plan_p$ (that has the form of a set of partial ordering constraints between tasks; it is empty if no failure has happened yet) built till that moment by the external planner and updates it with the new sets Start and Finish. Such

```
PROC Main()
    $\langle RealityChanged \land \neg Finished \rightarrow [\textbf{Monitor}()]\rangle$.
    $\langle Recovered \land \neg Finished \rightarrow [\textbf{UpdateProcess}()]\rangle$.
    $\langle \texttt{true} \rightarrow [\textbf{Process}(); finish]\rangle$.

PROC UpdateProcess()
    $receive(\textbf{Planner}, plan_h)$.
    $\textbf{Convert}(plan_h, \delta_h)$.
    $\textbf{Update}(\delta_0, (\delta_0 || \delta_h))$.
    $resetReality$.
    $resetRecovery$.

PROC Monitor()
    $\textbf{IndiPOP}()$.
    $resetRealityChanged$.

PROC IndiPOP()
    $Start = \Phi(s) \cup \{\bigcup_{i=1..k,j=1..n}\{Capable(c_i, t_j)\}\} \cup \{\bigcup_{i=1..k}\{Free(c_i, s)\}\}$.
    $Finish = \Psi(s)$.
    $receive(\textbf{Planner}, plan_p)$.
    $send(\textbf{RemoveConflicts}, [plan_p, Start, Finish])$.
    $receive(\textbf{RemoveConflicts}, plan_u)$.
    $send(\textbf{Planner}, [plan_u, Start, Finish])$.

PROC ManageExecution($Task, Input, ExpectedOutput$)
    $\pi(Srvc).assign(Srvc, Task, Input, ExpectedOutput)$.
    $start(Srvc, Task)$.
    $ackCompl(Srvc, Task)$.
    $release(Srvc, Task, Input, ExpectedOutput, RealOutput)$.
```

Fig. 3. A fragment of the core procedures of the IndiGolog PMS

updating finds something about $plan_p$ that needs fixing in according with the new realities. Since $plan_p$ has been built working on old values of the two realities, it is possible that some ordering constraints between tasks are not valid anymore. This causes the generation of some conflicts, that need to be deleted by $plan_p$ through the external procedure **RemoveConflicts**. Basically, **IndiPOP** can be seen as a conflict-removal procedure that revises the partial recovery plan to the new realities. At this point, $plan_u$ (that is, $plan_p$ just updated, i.e., without conflicts) is sent back to the external planner together with the sets Start and Finish. The external planner can now restore its planning procedure. Note that if the predicate Relevant(s) holds, meaning that a misalignment between the two realities exists, the PMS tries to continue with its execution. In particular, every T_i whose T-limited expected reality $\Psi|_{T_i}(s)$ is different from the T-limited physical reality $\Phi|_{T_i}(s)$ could not more proceed with its execution. However, every task T_j not affected by the deviation can advance without any obstacle. Once sent the sets of fluents composing the two realities to the external planner, the monitor resets the fluent *RealityChanged* to *false*, and the control passes to the process of interest (i.e., program **Process**), that may again execute/advance. When the external planner finds a recovery plan that can align physical and expected reality, the fluent *Recovered* is switched to *true* and the procedure **UpdateProcess** is enabled. Now, after receiving the recovery process δ_h from the planner, the PMS updates the original process δ_0 to a new process δ_0' that, respect to its predecessor, has a new branch to be executed in parallel; such branch is exactly δ_h. It contains all that tasks able to repair the physical reality from the discrepancies (i.e., to unblock all that tasks stopped in δ_0 because their preconditions did not hold). Note that when δ_h is merged with the original process δ_0, the two realities are still different from each others. Therefore, the PMS makes them equal by forcing $\Psi(s)$ to the current value of $\Phi(s)$. This because the purpose of δ_h, after that all recovery actions have been executed, is to turn the current $\Phi(s)$ into $\Psi(s')$, where s' is that situation reached after the execution of recovery actions. Let us now formalize the concept of *strongly consistency* for a process δ_0.

Definition 8 *Let δ_0 be a process composed by n tasks $T_1, .., T_n$. δ_0 is* ***strongly consistent*** *iff:*

- *Given a specific task T and an input I, $\nexists c, c', p, p', q, q', a, a'$ s.t. $a = release(c, T, I, p, q) \ \wedge a' = release(c', T, I, p', q') \wedge (p \neq p')$.*
- *$\forall j \in 1..m, \nexists (T_i, T_k)_{i \neq k}$ s.t.$(T_i \rhd X_{\varphi,j} \wedge T_k \rhd X_{\varphi,j})$.*

Intuitively, a process δ_0 is strongly consistent if a specific task, executed on a given input, cannot return different values for its expected output; moreover, the above condition holds if do not exist two different tasks that affect the same fluent. For strongly consistent processes, we can state the concept of *goal* :

Definition 9 *Given a strongly consistent process δ_0, composed by n tasks $T_1, ..., T_n$, the goal of δ_0 can be defined as the set of all expected fluents $X_{\psi,j}$ that are affected by $T_1, ..., T_n$. Hence, $Goal(\delta_0) = \{X_{\psi,j}$ s.t. $\exists i_{1..n}.(T_i \rhd X_{\psi,j})\}$.*

After a recovery procedure δ_h, $Goal(\delta_0) \subseteq Goal(\delta_0 || \delta_h)$, since the recovery procedure can introduce new tasks with respect to the original process δ_0. Anyway, the original $Goal(\delta_0)$ is preserved also after the adaptation procedure.

Theorem 1 (Termination). *Let δ_0 be a strongly consistent process composed by a finite number of tasks $T_1, ..., T_n$. If δ_0 does not contain while and iteration constructs (cf. Table 1), and the number of exogenous events is finite, then the core procedure of* IndiGolog *PMS terminates.*

We want to underline that the termination cannot be guaranteed if δ_0 contains loops or iterations, since potentially the two realities could indefinitely change. The same is true if the number of exogenous events is unbounded.

Example 3. Suppose that the process depicted in Figure 1 starts its execution and reaches a situation s where some tasks have been completed by returning their expected outputs. In particular, suppose that the left branch of the process has been completely executed, by obtaining $PhotoOK_{\varphi}(dest_A, s) = true$ and $PhotoOK_{\psi}(dest_A, s) = true$, whilst the other tasks are still under execution. We have defined an exogenous event $photoLost(d)$ where d is a specific location. Such an exogenous event models the case when some photos, previously taken in d, get lost (e.g., due to the unwilling deletion of some files). Consequently, if the exogenous event $photoLost(dest_A)$ occurs, its effect is to force $PhotoOK_{\varphi}(dest_A, \overline{s})$ in the new situation $\overline{s}$ to be *false*, whilst $PhotoOK_{\psi}(dest_A, \overline{s})$ continues to hold. This means that $\mathsf{Relevant}(\delta_0, \overline{s}) \equiv \neg\mathsf{SameState}(\Phi(\overline{s}), \Psi(\overline{s}))$ and that the PMS

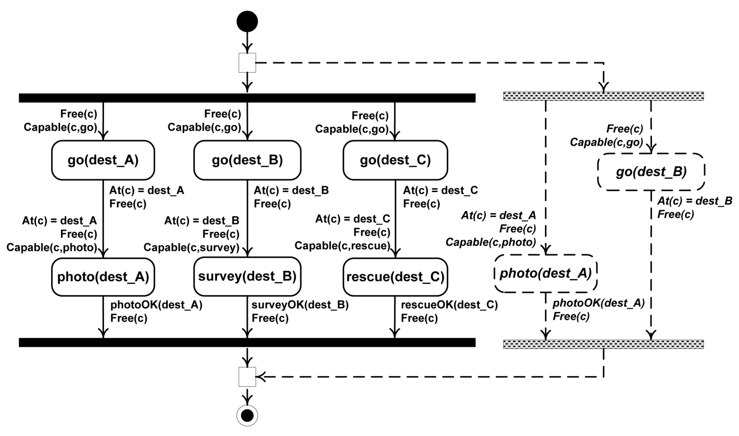


Fig. 4. The process in Figure 1 just fixed with a new repairing branch

should find a recovery program which restores the previous value for the fluent $PhotoOK_{\varphi}$. For this purpose, the PMS invokes the external planner. While the planner starts to build the recovery process, let us see the case in which, in the meanwhile, the task $go(dest_B)$ terminates with a different output by the expected one. In particular, suppose that the service $srvc_2$, that is executing $go(dest_B)$, reaches $dest_Z$ instead of $dest_B$. Hence, we have different values for $At_{\varphi}(srvc2, \overline{s'})$ and $At_{\psi}(srvc2, \overline{s'})$. Again, the PMS invokes the external planner by obtaining the partial plan $plan_p$ built till that moment and verifies if it needs to be fixed according with new values of the two realities. If no conflicts are individuated, the PMS sends back $plan_p$ to the planner together with the information about the initial state and the goal, updated to situation $\overline{s'}$. Note that in situation $\overline{s'}$ the task *survey* cannot proceed because one of its preconditions does not hold. When the planner ends its computation, it returns the recovery process δ_h, that can be executed in concurrency with δ_0 (see the right-hand side of Figure 4) by preserving its original goal. ■

7 Conclusions

In this paper, we advocated the use of a declarative model named SmartPM for automatic process adaptation based on continuous planning. If an unexpected deviation is detected, a recovery process will be built and executed in parallel to the main process. The non-blocking repairing technique enables to reach all goals that would have been reached in the original process. Future works will include an extensive validation of the approach with real collaborative processes and will face the drawbacks provided by the use of continuous planning, such as the risk to introduce data inconsistency when repairing.

References

1. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer-Verlag New York (2007)
2. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative Workflows: Balancing between Flexibility and Support. Computer Science - R&D 23, 99–113 (2009)
3. Brenner, M., Nebel, B.: Continual Planning and Acting in Dynamic Multiagent Environments. Autonomous Agents and Multi-Agent Systems 19, 297–331 (2009)
4. Gajewski, M., Meyer, H., Momotko, M., Schuschel, H., Weske, M.: Dynamic Failure Recovery of Generated Workflows. In: 16th Int. Conf. on Database and Expert Systems Applications. IEEE Computer Society Press, pp. 982–986 (2005)
5. R-Moreno, M.D., Borrajo, D., Cesta, A., Oddi, A.: Integrating Planning and Scheduling in Workflow Domains. Expert Syst. Appl. 33, 389–406 (2007)
6. Hofstede, A., van der Aalst, W., Adams, M., Russell, N.: Modern Business Process Automation: YAWL and its Support Environment. Springer Publ. Company (2009)
7. Hagen, C., Alonso, G.: Exception Handling in Workflow Management Systems. IEEE Trans. Soft. Eng. 26(10), 943–958 (2000)
8. Göser, K., Jurisch, M., Acker, H., Kreher, U., Lauer, M., Rinderle, S., Reichert, M., Dadam, P.: Next-generation Process Management with ADEPT2. In: Demonstration Prog. at the Int. Conf. on Business Process Management (2007)
9. Chiu, D., Li, Q., Karlapalem, K.: A Logical Framework for Exception Handling in ADOME Workflow Management System. In: 12th Int. Conf. of Advanced Information Systems Engineering, pp. 110–125 (2000)
10. Müller, R., Greiner, U., Rahm, E.: AGENTWORK: a Workflow System Supporting Rule-based Workflow Adaptation. Data & Knowledge Eng. 51(2), 223–256 (2004)
11. Weber, B., Reichert, M., Rinderle-Ma, S., Wild, W.: Providing Integrated Life Cycle Support in Process-Aware Information Systems. Int. J. of Cooperative Information Systems 18(1), 115–165 (2009)
12. Weske, M.: Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System. In: Hawaii Int. Conf. on System Sciences (2001)
13. Mcdermott, D., Ghallab, M., Howe, A., Knoblock, C.A., Ram, A., Veloso, M., Weld, D.S., Wilkins, D.E.: PDDL - The Planning Domain Definition Language. Technical report (1998)
14. Penberthy, S.J., Weld, D.S.: UCPOP: A Sound, Complete, Partial Order Planner for ADL. In: Int. Conf. on the Principles of Knowledge Representation and Reasoning, pp. 103–114 (1992)
15. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press (2001)
16. De Giacomo, G., Lespérance, Y., Levesque, H.J., Sardina, S.: IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents. In: Multi-Agent Programming, 31–72 (2009).
17. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.: On Structured Workflow Modelling. In: Int. Conf. of Advanced Information Systems Eng., pp. 431–445 (2000)
18. de Leoni, M., Mecella, M., De Giacomo, G.: Highly dynamic Adaptation in Process Management Systems through Execution Monitoring. In: 5th Int. Conf. on Business Process Management, pp. 182–197 (2007)
19. de Leoni, M., De Giacomo, G., Lespèrance, Y., Mecella, M.: On-line Adaptation of Sequential Mobile Processes Running Concurrently. In: ACM Symposium on Applied Computing, pp. 1345–1352 (2009)