Highly Dynamic Adaptation in Process Management Systems through Execution Monitoring

Massimiliano de Leoni, Massimo Mecella, and Giuseppe De Giacomo

Dipartimento di Informatica e Sistemistica
SAPIENZA – Università di Roma
Via Ariosto 25, 00185 Roma, Italy
{deleoni,mecella,degiacomo}@dis.uniroma1.it

Abstract. Nowadays, process management systems can be used not only in classical business scenarios, but also in highly mobile and dynamic situations, e.g., in supporting operators during emergency management in order to coordinate their activities. In such challenging situations, processes should be adapted, in order to cope with anomalous situations, including connection anomalies and task faults. In this paper, we present a general approach, based on execution monitoring, which is *(i)* practical, by relying on well-established planning techniques, and *(ii)* does not require the definition of the adaptation strategy in the process itself (as most of the current approaches do). We prove the correctness and completeness of the approach.

1 Introduction

Nowadays, process management systems (PMSs, [1, 2]) are widely used in many business scenarios, such as government agencies, insurances, banks, etc. Besides such scenarios, which present mainly static characteristics (i.e., deviations are not the rule, but the exception), PMSs can be used also in mobile and highly dynamic situations, such as in coordinating operators/devices/robots/sensors in emergency situations [3, 4].

As an example, in [5] a project is presented in which PMSs are used within teams of emergency operators, in order to coordinate their activities. In such scenarios, the members of a team are equipped with PDAs and coordinated through a PMS residing on a leader device (usually a laptop); devices communicate among them through ad hoc networks, and in order to carry on the process, they need to be continually connected each other. But this is not simply guaranteed: the environment is highly dynamic, since nodes (i.e., devices and the related operators) move in the affected area to carry out assigned tasks; movements may cause possible disconnections and, so, unavailability of nodes. Therefore the process should be adapted. Adaptivity might simply consist in assigning the task in progress to another device, but collecting actual user requirements [6] shows that typical teams are formed by a few nodes (less than 10 units), and therefore frequently such reassignment is not feasible. Conversely, other kind of adaptivity can be envisioned, such as recovering somehow the disconnecting node through

specific tasks, e.g., when X is disconnecting, the PMS could assign the "follow X" task to another node so to guarantee the connection. This example shows that in such scenarios *(i)* the process is designed (and deployed on the PMS) as if everything would be fine during run-time, and *(ii)* it needs to be continuously adapted on the basis of rules that would be infeasible to foresee at design time.

The aim of this paper is to propose a general conceptual framework for the above issue, and to present a practical technique for solving it, which is based on planning in AI; moreover, we prove the correctness and completeness of the approach. In a PMS, process schemas are defined that describe the different aspects, i.e., tasks/activities, control and data flow, tasks assignment to services [1], etc. Every task gets associated a set of conditions which have to be true in order to perform the task. Conditions are defined on control and data flow (e.g., a previous task has to be finished, a variable needs to be assigned a specific range of values, etc.). This kind of conditions can be somehow considered as "internal": they are handled internally by the PMS and, thus, easily controllable. Another type of conditions exist, that is the "external" ones: they depend on the environment where process instances are carried on. These conditions are more difficult to keep under control and a continuous *monitoring* to detect discrepancies is required. Indeed we can distinguish between a *physical reality* and a *virtual reality* [7]; the physical reality is the actual values of conditions, whereas the virtual reality is the model of reality that PMS uses in making deliberations. A PMS builds the virtual reality by assuming the effects of tasks/actions fill expectations (i.e., they modify correctly conditions) and no exogenous events break out, which are capable to modify conditions.

When the PMS realizes that one or more events caused the two kinds of reality to deviate, there are three possibilities to deal with such a discrepancy:

1. Ignoring deviations – this is, of course, not feasible in general, since the new situation might be such that the PMS is no more able to carry out the process instance.
2. Anticipating all possible discrepancies – the idea is to include in the process schema the actions to cope with each of such failures. As we discuss in Section 7, most PMSs use this approach. For simple and mainly static processes, this is feasible and valuable; but, especially in mobile and highly dynamic scenarios, it is quite impossible to take into account all exception cases.
3. Devising a general recovery method able to handle any kind of exogenous events – this can be seen as a `try-catch` approach, used in some programming languages such as Java. The process is defined as if exogenous actions cannot occur, that is everything runs fine (the `try` block). Whenever the execution monitor (i.e., the module intended for execution monitoring) detects discrepancies leading the process instance not to be terminable, the control flow moves to the `catch` block. The `catch` block activates the general recovery method to modify the old process P in a process P' so that P' can terminate in the new environment and its goals are included in those of P.

[1] In this work, we abstract all possible actors a process can coordinate, i.e., human operators commonly interacting through worklists, software applications/components, etc. as *services* providing capabilities to be matched with the ones required by the tasks.

Here the challenge is to *automatically* synthesize P' during the execution itself, without specifying a-priori all the possible `catch`es.

The contribution of this paper is *(i)* to introduce a general conceptual framework in accordance with the third approach previously described, and *(ii)* to present a practical technique, in the context of this framework, that is able to *automatically* cope with anomalies. We prove the correctness and completeness of such a technique, which is based on planning techniques in AI.

The rest of the paper is organized as follows: Section 2 introduces some preliminary notions, namely Situation Calculus and ConGolog, that are used as proper formalisms to reason about processes and exogenous events. Section 3 presents the general conceptual framework to address adaptivity in highly dynamic scenarios, and introduces a running example. Section 4 presents the proposed formalization of processes, and Section 5 deals with the adaptiveness. Section 6 presents the specific technique and proves its correctness and completeness. Related works are discussed in Section 7, and finally Section 8 concludes the paper.

2 Preliminaries

In this section we introduce the Situation Calculus, which we use to formalize the adaptiveness in PMSs. The Situation Calculus [8] is a second-order logic targeted specifically for representing a dynamically changing domain of interest (the world). All changes in the world are obtained as result of *actions*. A possible history of the actions is represented by a *situation*, which is a first-order term denoting the current situation of the world. The constant s_0 denotes the initial situation. A special binary function symbol $do(\alpha, s)$ denotes the next situation after performing the action α in the situation s. Action may be parameterized.

Properties that hold in a situation are called *fluents*. These are predicates taking a situation term as their last argument. 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:

$$F(\overrightarrow{x}, do(\alpha, s)) \Leftrightarrow \Phi_F(\overrightarrow{x}, do(\alpha, s), s)$$

where $\Phi_F(\overrightarrow{x}, do(\alpha, s), s)$ is a formula with free variables $\overrightarrow{x}$, α is an action, and s is a situation. Besides successor state axioms, Situation Calculus theories are characterized by *action precondition axioms*, which specify whether a certain action is executable in a situation. Action precondition axioms have the form:

$$Poss(\alpha, s) \Leftrightarrow \Pi_\alpha(s)$$

where the formula $\Pi_\alpha(s)$ defines the conditions under which the action α may be performed in the situation s.

In order to control the executions of actions we make use of high level programs, expressed in Golog-like programming languages [9]. In particular we focus on ConGolog [10] which is equipped with primitives for expressing concurrency.

Construct	**Meaning**
a	A primitive action
$\phi?$	Wait while the ϕ 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 procedure passing a vector $\overrightarrow{v}$ of parameters
$if\ \phi\ then\ \delta_1\ else\ \delta_2$	Exclusive choice between δ_1 and δ_2 according to the condition ϕ
$while\ \phi\ do\ \delta$	Iterative invocation of δ
$(\delta_1 \parallel \delta_2)$	Concurrent execution

Table 1. ConGolog constructs

The Table 1 summarizes the constructs of ConGolog used in this work. Basically, these constructs allow to define every well-structured process as defined in [11].

From the formal point of view, ConGolog programs are terms. The execution of ConGolog programs is expressed through a *transition semantic* based on single steps of execution. At each step a program executes an action and evolves to a new program which represents what remains to be executed of the original program. Formally two predicates are introduced to specify such a sematic:

- $Trans(\delta', s', \delta'', s'')$, given a program δ' and a situation s', returns *(i)* a new situation s'' resulting from executing a single step of δ', and *(ii)* δ'' which is the remaining program to be executed.
- $Final(\delta', s')$ returns true when the program δ' can be considered successfully completed in situation s'.

By using $Trans$ and $Final$ we can define a predicate $Do(\delta', s', s'')$ that represent successful complete executions of a program δ' in a situation s', where s'' is the situation at the end of the execution of δ'. Formally:

$$Do(\delta', s', s'') \Leftrightarrow \exists \delta''. Trans^*(\delta', s', \delta'', s'') \wedge Final(\delta'', s'')$$

where $Trans^*$ is the definition of the reflective and transitive closure of *Trans*.

3 General Framework

The general framework which we introduce in this paper is based on execution monitoring formally represented in Situation Calculus [12, 7]. After each action, the PMS has to align the internal world representation (i.e., the virtual reality) with the external one (i.e., the physical reality), since they could differ due to unforeseen events.

When using ConGolog for process management, tasks are considered as predefined sequences of actions (see later) and processes as ConGolog programs.

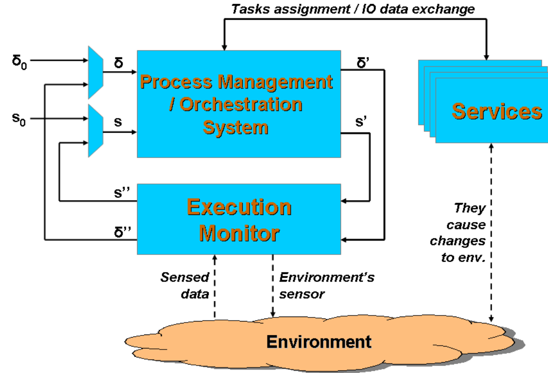

Fig. 1. Execution Monitoring

Before a process starts to be executed, the PMS takes the initial context from the real environment as initial situation, together with the program (i.e. the process) δ_0 to be carried on. The initial situation s_0 is given by first-order logic predicates. For each execution step, the PMS, which has a complete knowledge of the internal world (i.e., its virtual reality), assigns a task to a service. The only assignable tasks are those ones whose preconditions are fulfilled. A service can collect from the PMS the data which are required in order to execute the task. When a service finishes executing the task, it alerts the PMS of its completion.

The execution of the PMS can be interrupted by the monitor when a misalignment between the virtual and the physical reality is sensed. When this happens, the monitor adapts the program to deal with such a discrepancy.

Figure 1 illustrates such an execution monitoring. At each step, PMS advances the process δ in the situation s by executing an action, resulting in a new situation s' with the process δ' remaining to be executed. The state[2] is represented as first-order formulas that are defined on situations. The current state corresponds to the boolean values of these formulas evaluated on the current situation.

Both the situation s' and the process δ' are given as input to the monitor. It collects data from the environment through *sensors* (here *sensor* is any software or hardware component enabling to retrieve contextual information). If a discrepancy between the virtual reality as represented by s' and the physical reality is sensed, the monitor changes s' in s'' by internally simulating a sequence of actions that re-aligns the virtual and physical reality (i.e., those are not really executed). Notice that the process δ' may fail to be correctly executed (i.e., by assigning all tasks as required) in s''. If so, the monitor adapts the process by generating a new process δ'' that pursues at least each δ''s goal and is executable

[2] Here we refer as *state* both the tasks' state (e.g, performable, running, terminated, etc.) and the process' variables. The use of the latter variables are twofold: from the one hand, the routing is defined on them and, from the other hand, they allow to learn when a task may fire.

in s''. At this point, the PMS is resumed and the execution is continued from δ'' and s''.

We end this section by introducing our running example, stemming from the project described in [5, 6].

Example 1 *A Mobile Ad hoc NETwork (*MANET*) is a P2P network of mobile nodes capable of communicating with each other without an underlying infrastructure. Nodes can communicate with their own neighbors (i.e., nodes in radio-range) directly by wireless links. Non-neighbor nodes can communicate as well, by using other intermediate nodes as relays that forward packets toward destinations. The lack of a fixed infrastructure makes this kind of network suitable in all scenarios where it is needed to deploy quickly a network, but the presence of access points is not guaranteed, as in emergency management.*

Coordination and data exchange requires MANET *nodes to be continually connected each other. But this is not guaranteed in a* MANET*. The environment is highly dynamic, since nodes move in the affected area to carry out assigned tasks. Movements may cause possible disconnections and, so, unavailability of nodes, and, consequently, unavailability of provided services. Therefore processes should be adapted, not simply by assigning tasks in progress to other services, but also considering possible recovery of the services.*

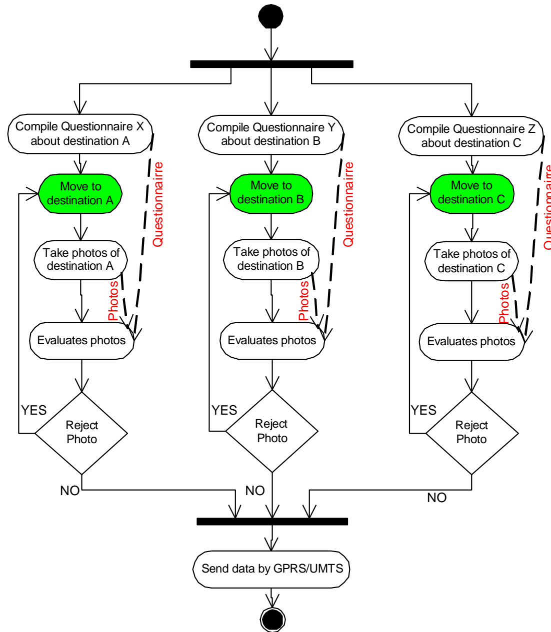

Fig. 2. A possible process to be carried on in disaster management scenarios

Figure 2 shows a possible scenario for information collecting after an earthquake: a team is sent to the affected area to evaluate the situation of three buildings. For each building, an actor compiles a questionnaire (by using a service, i.e., an application that it has got installed). Questionnaire compiling can be done everywhere: that is, movement is not required. Then, another actor/service has to be sent to the specific building to collect some pictures (this, conversely, requires movement). Finally, according to information in the questionnaire, a third actor/service evaluates quality and effectiveness of collected pictures. If pictures are of bad quality, the task of taking new pictures is scheduled again. Whenever these steps have been performed for the three buildings A, B and C, the collected data (questionnaires and pictures) are sent by GPRS or UMTS elsewhere. □

4 Formalization in Situation Calculus

Next we detail the general framework proposed above by using Situation Calculus and ConGolog. We use some domain-independent predicates to denote the various objects of interest in the framework:

- $service(a)$: a is a service
- $task(x)$: x is a task
- $capability(b)$: b is a capability
- $provide(a, b)$: the service a provides the capability b
- $require(x, b)$: the task x requires the capability b

Every task execution is the sequence of four actions: *(i)* the assignment of the task to a service, resulting in the service being not free anymore; *(ii)* the notification to the service to start executing the task. Then, the service carries out the tasks and, after finishing, *(iii)* the PMS stops the service, acknowledging the successful termination of its task. Finally, *(iv)* the PMS releases the service, which becomes free again. We formalize these four actions as follows (these are the only actions used in our formalization):

- $Assign(a, x)$: the task x is assigned to a service a
- $Start(a, x, p)$: the service a is notified to perform the task x on input p
- $Stop(a, x, q)$: the service a is stopped acknowledging the successful termination of x with output q
- $Release(a, x)$: the service a is released with respect to the task x

The terms p and q denote arbitrary sets of input/output, which depend on the specific task; if no input or output is needed, p and q are $\emptyset$.

For each specific domain, we have several fluents representing the properties of situations. Among them, we have the fluent $free(a, s)$, which is indeed domain-independent, that denotes the fact that the service a is free, i.e., no task has been assigned to it, in the situation s. The corresponding successor state axiom is as follows:

$$\begin{aligned} & free(a, do(t, s)) \Leftrightarrow \\ & \quad \big(\forall x. t \neq Assign(a, x) \wedge free(a, s)\big) \vee \\ & \quad \big(\neg free(a, s) \wedge \exists x. t = Release(a, x)\big) \end{aligned} \tag{1}$$

This says that a service a is considered free in the current situation if and only if a was free in the previous situation and no tasks have been just assigned to it, or a was not free and it has been just released.

In addition, we make use, in every specific domain, of a predicate $available(a, s)$ which denotes whether a service a is available in situation s for tasks assignment. However, *available* is domain-dependent and, hence, requires to be defined specifically for every domain. Its definition must enforce the following condition:

$$\forall a\ s.available(a, s) \Rightarrow free(a, s) \tag{2}$$

Precondition axioms are also domain dependent and, hence, vary from case to case. However the following condition must be true:

$$\forall x, p, q\ Poss\big(Start(a, x, p), s\big) \Rightarrow available(a, s) \tag{3}$$

Knowing whether a service is available is very important for the PMS when it has to perform assignments. Indeed, a task x is assigned to the best service a which is available and provides every capability required by x. In order to model the best choice among available services, we introduced a special construct, named *pick*. The statement $pick\ a.[\phi(a)]\ \delta$ chooses the best service a matching the condition $\phi(\cdot)$ on predicates and fluents applied in the current situation. This choice is performed with respect to a given (typically domain-dependent) metric. For instance, the *pick* might consider the queueing of tasks that are assigned to the members, as well as the execution of multiple parallel process instances sharing the same resources. Observe that such a choice is deterministic, even when there are more than one service matching the condition $\phi(\cdot)$. Then it instantiates δ with the chosen a and executes the first step of the resulting program. If no service matches the condition, the process δ stays blocked, until some other services make $\phi(a)$ true.

We illustrate such notions on our running example.

Example 1 (cont.). *We formalize the scenario in Example 1. We make use of the following domain-dependent fluents (for sake of brevity we focus on the most relevant ones):*

- $connected(a, b, s)$*: which is true if in the situation s the services a and b are connected through multi-hop paths*
- $neigh(a, b, s)$*: which is true if in the situation s the services a and b are in radio-range in the situation s*
- $at(a, p, s)$ *it is true if in the situation s the service a is located at the coordinate $p = \langle p_x, p_y, p_z \rangle$ in the situation s.*

The successor state axioms for this domain are:

$$available(a, do(x, s)) \Leftrightarrow free(a, do(x, s)) \wedge connected(a, \texttt{Coord}, do(x, s))$$

$$connected(a_0, a_1, do(x, s)) \Leftrightarrow neigh(a_0, a_1, do(x, s)) \vee \big(\exists a_2.service(a_2) \wedge neigh(a_0, a_2, do(x, s)) \wedge connected(a_2, a_1, do(x, s))\big)$$

$$neigh(a_0, a_1, do(x, s)) \Leftrightarrow at(a_0, p_0, do(x, s)) \wedge at(a_1, p_1, s) \wedge \parallel p_0 - p_1 \parallel < rrange$$

$$at(a, p, do(x, s)) \Leftrightarrow \forall\ p, q.\ x \neq Stop(a, \texttt{Go}, p) \wedge at(a, p, s) \vee\ \forall\ q.\ x = Stop(a, \texttt{Go}, p)$$

The first successor state axiom is the one for `available`*, which states a service is available if it is connected to the coordinator device (denoted by* `Coord`*) and it is free. Notice that the condition 2 is fulfilled. The axiom for* `connected` *states two devices are connected if and only if they are neighbors (i.e., in radio-range) or there exists a path in the* MANET*. The successor state axiom* `neigh` *states how neighbors evolve: two nodes a and b are neighbors in situation s if and only if their distance* $\parallel p_a - p_b \parallel$ *is less than the radio-range. The successor state axiom for* `at` *states that the position p for a does not change until the assigned task* `Go` *is acknowledged to be finished.*

In the reality, in order to know the position returned by the `Go` *task, the PMS gets such an information from the service a (here not modelled). In case the node (hence the service) is getting disconnected, the monitor will get in and generate a* `Stop` *action which communicates the actual position, and a recovery is instructed in order to keep the connection (see later). Indeed, in such a scenario nodes need be continually connected to each other, as a disconnected node is out of the PMS's control [3].*

For sake of brevity we do not look at the precondition axioms, and instead we look directly at the ConGolog *program (implicitly assuming that such precondition axioms allow for all instructions in the program). The* ConGolog *program corresponding to Figure 2 is shown in Figure 3. The main program is the procedure* Process*, which executes in parallel on three threads the sub-procedure* EvalTake *and then assigns the task* SendByGPRS *to the proper service that is the one providing the capability to send data by means of GPRS (or similar).* □

5 Adaptation

Next we formalize how the monitor works. Intuitively, the monitor takes the current program δ' and the current situation s' from the PMS's virtual reality and, analyzing the physical reality by sensors, introduces fake actions in order to get a new situation s'' which aligns the virtual reality of the PMS with sensed information. Then, it analyzes whether δ' can still be executed in s'', and if not, it adapts δ' by generating a new correctly executable program δ''. Specifically, the monitor work can be abstractly defined as follows (we do not model how the situation s'' is generated from the sensed information):

```
01 proc EvalTake(Location, Questionnaire, Photos)
02 pick a_o [actor(a_0) ∧ available(a_0) ∧ ∀b service(b) ∧ require(b, Compile) ⇒ provide(a_0, b)]
03          Assign(a_0, Compile);
04          Start(a_0, Compile, Location);
05          Stop(a_0, Compile, Questionnaire);
06          Release(a_0, Compile);
07 isOk := false;
08 while(isOk == false)
09          (pick a_1[actor(a_1) ∧ available(a_1) ∧ ∀b service(b) ∧ require(b, TakePhoto)
            ⇒ provide(a_1, b)]
10                   Assign(a_1, TakePhoto);
11                   Start(a_1, Go, Location);
12                   Stop(a_1, Go, Location);
13                   Start(a_1, TakePhoto, Location);
14                   Stop(a_1, TakePhoto, Photos);
15                   Release(a_1, TakePhoto); )
16          pick a_2[actor(a_2) ∧ available(a_2) ∧ ∀b service(b) ∧ require(b, Evaluate)
            ⇒ provide(a_2, b)]
17                   Assign(a_2, Evaluate);
18                   Start(a_1, Evaluate, {Location, Questionnaire, Photos});
19                   Stop(a_1, Evaluate, isOk);
20                   Release(a_2, Evaluate); )
21 endproc
22
23 proc Process
24 (EvalTake(LocA, Q_a, F_a) ||
25 EvalTake(LocB, Q_b, F_b) ||
26 EvalTake(LocC, Q_c, F_c));
27 pick a[actor(a)] ∧ available(a) ∧ ∀b service(b) ∧ require(b, SendByGPRS)
   ⇒ provide(a, b)?;
28          Assign(a, SendByGPRS);
29          Start(a, SendByGPRS, {Q_a, F_a, Q_b, F_b, Q_c, F_c});
30          Stop(a, SendByGPRS, nil);
31          Release(a, SendByGPRS);
32 endproc
```

Fig. 3. The ConGolog program of the process in Figure 2

$$\begin{aligned}&Monitor(\delta', s', s'', \delta'') \Leftrightarrow\\&\quad \big(Relevant(\delta', s', s'') \wedge Recovery(\delta', s', s'', \delta'')\big) \vee\\&\quad \big(\neg Relevant(\delta', s', s'') \wedge \delta'' = \delta'\big)\end{aligned} \tag{4}$$

where: *(i)* $Relevant(\delta', s', s'')$ states whether the change from the situation s' into s'' is such that δ' cannot be correctly executed anymore; and *(ii)* $Recovery(\delta', s', s'', \delta'')$ is intended to hold whenever the program δ', to be originally executed in the situation s', is adapted to δ'' in order to be executed in the situation s''.

Formally *Relevant* is defined as follows:

$$Relevant(\delta', s', s'') \Leftrightarrow \neg SameConfig(\delta', s', \delta', s'')$$

where $SameConfig(\delta', s', \delta'', s'')$ is true if executing δ' in s' is "equivalent" to executing δ'' in s'' (see later for further details).

In this general framework we do not give a definition for $SameConfig(\delta', s', \delta'', s'')$. However we consider any definition for $SameConfig$ to be correct if it denotes a bisimulation [13]. Formally, for every $\delta', s', \delta'', s''$ holds:

1. $Final(\delta', s') \Leftrightarrow Final(\delta'', s')$
2. $\forall\ a, \delta'.Trans\big(\delta', s', \overline{\delta'}, do(a, s')\big) \Rightarrow$
   $\exists\ \overline{\delta''}.Trans\big(\delta'', s'', \overline{\delta'}, do(a, s'')\big) \wedge SameConfig\big(\overline{\delta'}, do(a, s), \overline{\delta''}, do(a, s'')\big)$
3. $\forall\ a, \delta'.Trans\big(\delta'', s'', \overline{\delta'}, do(a, s'')\big) \Rightarrow$
   $\exists\ \overline{\delta''}.Trans\big(\delta', s', \overline{\delta'}, do(a, s')\big) \wedge SameConfig\big(\overline{\delta''}, do(a, s''), \overline{\delta'}, do(a, s')\big)$

Intuitively, a predicate $SameConfig(\delta', s', \delta'', s'')$ is said to be correct if δ' and δ'' are terminable either both or none of them. Furthermore, for each action a performable by δ' in the situation s', δ'' in the situation s'' has to enable the performance of the same actions (and viceversa). Moreover, the resulting configurations $(\overline{\delta'}, do(a, s'))$ and $(\overline{\delta''}, do(a, s'))$ must still satisfy $SameConfig$.

The use of the bisimulation criteria to state when a predicate $SameConfig(\cdots)$ is correct, derives from the notion of equivalence introduced in [14]. When comparing the execution of two formally different business processes, the internal states of the processes may be ignored, because what really matters is the process behavior that can be observed. This view reflects the way a PMS works: indeed what is of interest is the set of tasks that the PMS offers to its environment, in response to the inputs that the environment provides.

Next we turn our attention to the procedure to adapt the process formalized by $Recovery(\delta, s, s', \delta')$. Formally is defined as follows:

$$\begin{aligned}&Recovery(\delta', s', s'', \delta'') \Leftrightarrow\\&\quad \exists \delta_a, \delta_b.\delta'' = \delta_a; \delta_b \wedge Deterministic(\delta_a) \wedge\\&\quad Do(\delta_a, s'', s_b) \wedge SameConfig(\delta', s', \delta_b, s_b)\end{aligned} \tag{5}$$

Recovery determines a process δ'' consisting of a *deterministic* δ_a (i.e., a program not using the concurrency construct), and an arbitrary program δ_b. The aim of δ_a is to lead from the situation s'' in which adaptation is needed to a new situation s_b where $SameConfig(\delta', s', \delta_b, s_b)$ is true.

Notice that during the actual recovery phase δ_a we disallow for concurrency because we need full control on the execution of each service in order to get to a recovered state. Then the actual recovered program δ_b can again allow for concurrency.

6 Adaptation: A Specific Technique

In the previous sections we have provided a general description on how adaptation can be defined and performed. Here we choose a specific technique that is actually feasible in practice. Our main step is to adopt a specific definition for $SameConfig$, here denoted as SAMECONFIG, namely:

$$\begin{aligned} &\textsc{SameConfig}(\delta', s', \delta'', s'') \Leftrightarrow \\ &\quad SameState(s', s'') \wedge \delta' = \delta'' \end{aligned} \tag{6}$$

In other words, SAMECONFIG states that δ', s' and δ'', s'' are the same configuration if *(i)* all fluents have the same truth values in both s' and s'' ($SameState$)[3] , and *(ii)* δ'' is actually δ'.

The following shows that SAMECONFIG is indeed correct.

Theorem 1. $\textsc{SameConfig}(\delta', s', \delta'', s'')$ *is correct.*

Proof. We show that SAMECONFIG is a bisimulation. Indeed:

- Since $SameState(s', s'')$ requires all fluents to have the same values both in s' and s'', we have that $\big(Final(\delta, s') \Leftrightarrow Final(\delta, s'')\big)$.
- Since $SameState(s', s'')$ requires all fluents to have the same values both in s' and s'', it follows that the PMS is allowed for the same process δ' to assign the same tasks both in s' and in s'' and moreover for each action a and situation s' and s'' s.t. $SameState(s', s'')$, we have that $SameState(do(a, s'), do(a, s''))$ hold. As a result, for each a and $\overline{\delta'}$ such that $Trans\big(\delta', s', \overline{\delta'}, do(a, s')\big)$ we have that $Trans\big(\delta', s'', \overline{\delta'}, do(a, s'')\big)$ and $\textsc{SameConfig}\big(\overline{\delta'}, do(a, s), \overline{\delta''}, do(a, s'')\big)$. Similarly for the other direction.

Hence, the thesis holds. □

Next let us denote by $LinearProgram(\delta)$ a program constituted only by sequences of actions, and let us define RECOVERY as:

$$\begin{aligned} &\textsc{Recovery}(\delta', s', s'', \delta'') \Leftrightarrow \\ &\quad \exists \delta_a, \delta_b . \delta'' = \delta_a; \delta_b \wedge LinearProgram(\delta_a) \wedge \\ &\quad Do(\delta_a, s'', s_b) \wedge \textsc{SameConfig}(\delta', s', \delta_b, s_b) \end{aligned} \tag{7}$$

Next theorem shows that we can adopt RECOVERY as a definition of $Recovery$ without loss of generality.

[3] Observe that $SameState$ can actually be defined as a first-order formula over the fluents, as the conjunction of $F(s') \Leftrightarrow F(s'')$ for each fluent F.

Theorem 2. *For every process δ' and situations s' and s'', there exists a δ'' such that* $\text{RECOVERY}(\delta', s', s'', \delta'')$ *if and only if there exists a $\overline{\delta''}$ such that $Recovery(\delta', s', s'', \overline{\delta''})$, where in the latter we use* SAMECONFIG *as $SameConfig$.*

Proof. Observe that the only difference between the two definitions is that in one case we allow only for linear programs (i.e., sequences of actions) as δ_a, while in the second case also for deterministic ones, that may include also `if-then-else`, `while`, procedures, etc.
($\Rightarrow$) Trivial, as linear programs are deterministic programs.
($\Leftarrow$) Let us consider the recovery process $\overline{\delta''} = \delta_a; \delta_b$ where δ_a is an arbitrary deterministic program. Then by definition of *Recovery* there exists a (unique) situation s'' such that $Do(\delta_a, s', s'')$. Now consider that s'' as the form $s'' = do(a_n, do(a_{n-1}, \ldots, do(a_2, do(a_1, s')) \ldots))$. Let us consider the linear program $p = (a_1; a_2; \ldots; a_n)$. Obviously we have $Do(p, s', s'')$. Hence the process $\delta'' = p; \delta_b$ is a recovery process according to the definition of RECOVERY. □

The nice feature of RECOVERY is that it asks to search for a linear program that achieves a certain formula, namely $SameState(s', s'')$. That is we have reduced the synthesis of a recovery program to a classical Planning problem in AI [15]. As a result we can adopt a well-developed literature about planning for our aim. In particular, if the services and input and output parameters are finite, then the recovery can be reduced to *propositional* planning, which is known to be decidable in general (for which very well performing software tools exists).

Theorem 3. *Let assume a domain in which services and input and output parameters are finite. Then given a process δ' and situations s' and s'', it is decidable to compute a recovery process δ'' such that* $\text{RECOVERY}(\delta', s', s'', \delta'')$ *holds.*

Proof. In domains in which services and input and output parameters are finite, also actions and fluents instantiated with all possible parameters are finite. Hence we can phrase the domain as a propositional one and the thesis follows from decidability of propositional planning [15]. □

Example 1 (cont.). *In the running example, consider the case in which the process is between the lines 11 and 12 in the execution of the procedure invocation $EvalTake(\texttt{LocA}, \texttt{Q_A}, \texttt{F_A})$. Now, let us assume that the node a_1 is assigned the task `TakePhoto`. But it is moving to a location such that it is not connected to the coordinator anymore; the monitor sees that it is getting out of reach and generates a spurious (not inside the original process) action $Stop(\texttt{a_1}, \texttt{Go}, \texttt{RealPosition})$, where `RealPosition` is the actual position as sensed by the monitor. Since `RealPosition` is not `LocA`, adaptation is needed; the Monitor generates the recovery program $\delta_a; \delta_b$ where δ_b is the original one from line 11 and δ_a is as follows:*

$$
\begin{array}{l}
Start(\texttt{a_3}, \texttt{Go}, \texttt{NewLocation});\\
Stop(\texttt{a_3}, \texttt{Go}, \texttt{NewLocation});\\
Start(\texttt{a_1}, \texttt{Go}, \texttt{LocA})
\end{array}
$$

	Schema	Instance		
		Pre-planned/Automatic	Ad hoc	
			Manual	Automatic
Woflan	Yes	No	Yes	No
ADEPT	Yes	No	Yes	No
WASA$_2$	Yes	No	Yes	No
Chautauqua	Yes	No	Yes	No
TRAM	Yes	No	Yes	No
Breeze	Yes	No	Yes	No
MILANO	Yes	No	No	No
WIDE	Yes	Yes	No	No
AgentWork	Yes	Yes	No	No
DYNAMITE	Yes	Yes	No	No
EPOS	Yes	Yes	No	No
MQ Worflow	Yes	No	No	No
Staffware	Yes	No	No	No
InConcert	Yes	No	Yes	No
SER Process	Yes	No	Yes	No
FileNet	Yes	No	Yes	No
FLOWer	Yes	No	Yes	No

Table 2. Comparison of process adaptation approaches present in literature.

where `NewLocation` *is within the radio-range of* `RealPosition`[4]. □

7 Related Works

Adaptation in PMSs can be considered at two level: at the process schema or at the process instance level [16]. Process schema changes become necessary, for example, to adapt the PMS to optimized business processes or to new laws [17–19]. In particular, applications supporting long-running processes (e.g., handling of mortgage or medical treatments) and the process instances controlled by them are affected by such changes. As opposed to this, changes of single process instances (e.g., to insert, delete, or shift single process steps) often have to be carried out in an ad-hoc manner in order to deal with an exceptional situation, e.g., peer disconnection in mobile networks [3], or evolving process requirements [20].

Table 2 shows a comparison of PMSs approaches supporting changes. The columns show the adaptation features addressed by existing PMSs. The second column illustrates which softwares support adaptation at schema level. As we can see, all analyzed softwares support it. The other three columns show the support for adaptation of single instances. The "Pre-Planned" PMSs refer to those systems that enable to specify adaptation rules to handle a set of exceptional (but foreseen) events. Conversely, the "Ad-hoc" support means PMSs to be able to adapt when unforeseen events fire. Ad-hoc adaptation of a process instance can be performed by the responsible person who manually changes the structure or, automatically, by the PMS.

[4] Observe that if the positions are discretized, so as to become finite, this recovery can be achieved by a propositional planner.

We note that there is no row having value "yes" in the column *Ad-hoc/Automatic*. That means that no considered approach allows users to manage unforeseen exceptions in a fully automatic way. Actually, only few systems (AgentWork[21], DYNAMITE[22], EPOS[23] and WIDE[24]) support automated process instance changes, but only in pre-planned way.

The work [25] is one of the few coping with exogenous events in the field of the Web service composition. This work considers the issue of long term optimality of the adaptation but, anyway, it does not manage unforeseen events. Moreover, it does require the definition of the probability according to which each of such events fires.

We underline that our approach is not another way to capture expected exceptions. Other approaches rely on rules to define the behaviors when special events are triggered. Here we simply model (a subset of) the running environment and the actions' effects, without considering possible special exceptional events. We argue that in some cases modeling the environment, even in detail, is easier than modeling all possible exceptions.

8 Conclusion

In this paper, we have presented a general approach, based on execution monitoring, for automatic process adaptation in dynamic scenarios. Such an approach is *(i)* practical, by relying on well-established planning techniques, and *(ii)* does not require the definition of the adaptation strategy in the process itself (as most of the current approaches do). We have proved the correctness and completeness of the approach, and we have shown its applicability to a running example stemming from a real project. Future works include to actually develop the Adaptive Process Management System. This will be done by using the `IndiGolog` module developed by the Cognitive Robotics Group of the Toronto University.

Acknowledgements. This work has been supported by the European Commission through the project FP6-2005-IST-5-034749 WORKPAD.

References

1. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice Hall PTR (1999)
2. van der Aalst, W., van Hee, K.: Workflow Management. Models, Methods, and Systems. MIT Press (2004)
3. de Leoni, M., Mecella, M., Russo, R.: A Bayesian Approach for Disconnection Management in Mobile Ad-hoc Networks. In: Proc. 4th International Workshop on Interdisciplinary Aspects of Coordination Applied to Pervasive Environments: Models and Applications (CoMA) (at WETICE 2007). (2007)
4. de Leoni, M., De Rosa, F., Mecella, M.: MOBIDIS: A Pervasive Architecture for Emergency Management. In: Proc. 4th International Workshop on Distributed and Mobile Collaboration (DMC 2006) (at WETICE 2006). (2006)
5. Catarci, T., De Rosa, F., de Leoni, M., Mecella, M., Angelaccio, M., Dustdar, S., Krek, A., Vetere, G., Zalis, Z.M., Gonzalvez, B., Iiritano, G.: WORKPAD:

2-Layered Peer-to-Peer for Emergency Management through Adaptive Processes. In: Proc. 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2006). (2006)
6. de Leoni, M., De Rosa, F., Marrella, A., Poggi, A., Krek, A., Manti, F.: Emergency Management: from User Requirements to a Flexible P2P Architecture. In: Proc. 4th International Conference on Information Systems for Crisis Response and Management (ISCRAM 2007). (2007)
7. De Giacomo, G., Reiter, R., Soutchanski, M.: Execution monitoring of high-level robot programs. In: KR. (1998) 453–465
8. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press (2001)
9. J. Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: Golog: A logic programming language for dynamic domains. J. Log. Program. **31** (1997) 59–83
10. De Giacomo, G., Lespérance, Y., J. Levesque, H.: Congolog, a concurrent programming language based on the situation calculus. Artif. Intell. **121** (2000) 109–169
11. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.: On structured workflow modelling. In: CAiSE. (2000) 431–445
12. Lesperance, Y., Ng, H.: Integrating planning into reactive high-level robot programs (2000)
13. Milner, R.: A Calculus of Communicating Systems. Springer (1980)
14. Hidders, J., Dumas, M., van der Aalst, W.M.P., ter Hofstede, A., Verelst, J.: When are two workflows the same? In: CATS. (2005) 3–11
15. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann (2004)
16. Weber, M., Illmann, T., Schmidt, A.: Webflow: Decentralized workflow management in the world wide web. In: Proc. of the International Conf. on Applied Informatics (AI'98). (1998)
17. Reichert, M., Rinderle, S., Dadam, P.: On the common support of workflow type and instance changes under correctness constraints. In: Proc. of the International Conf. on Cooperative Information Systems (CoopIS'03). (2003)
18. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems - a survey. Data and Knowledge Engineering **50** (2004) 9–34
19. van der Aalst, W., Basten, T.: Inheritance of Workflows: an approach to tackling problems related to change. Theoretical Computer Science, Elsevier Science Publishers **270** (2002) 125–203
20. Reichert, M., Dadam, P.: ADEPT$_{flex}$ supporting dynamic changes of workflows without losing control. Journal of Intelligent Information Systems **10** (1998) 93–129
21. Müller, R., Greiner, U., Rahm, E.: AGENTWORK: A Workflow-System Supporting Rule-Based Workflow Adaptation. Data and Knowledge Engineering **51(2)** (2004) 223–256
22. Heimann, P., Joeris, G., Krapp, C., Westfechtel, B.: DYNAMITE: dynamic task nets for software process management. In: Proc. of the 18th International Conference Software Engineering (ICSE). (Berlin, 1996) 331–341
23. Liu, C., Conradi, R.: Automatic replanning of task networks for process model evolution. In: Proc. of the European Software Engineers Conference. (Garmisch-Partenkirchen, Germany, 1993) 434–450
24. Ceri, S., Paul, Sanchez, G.: Wide: A distributed architecture for workflow management. In: 7th International Workshop on Research Issues in Data Engineering (RIDE). (1997)
25. Verma, K., Doshi, P., Gomadam, K., Miller, J., , Sheth, A.: Optimal adaptation in web processes with coordination constraints. In: The IEEE International Conference on Web Services (ICWS'06). (2006)