On-line Adaptation of Sequential Mobile Processes Running Concurrently

Massimiliano de Leoni[1]
Giuseppe De Giacomo[1] Yves Lespèrance[2] Massimo Mecella[1]
[1]Dipartimento di Informatica e Sistemistica, SAPIENZA - Universita' di Roma, Rome, Italy
{degiacomo,deleoni,mecella}@dis.uniroma1.it
[2]Department of Computer Science and Engineering, York University, Toronto, Canada
lesperan@cse.yorku.ca

ABSTRACT

Process Management Systems (PMSs) are nowadays more and more used as a supporting tool for cooperative processes in pervasive and highly dynamic situations, such as emergency situations, pervasive healthcare or domotics/home automation. But in all such situations, designed processes can be easily invalidated since the execution environment may change continuously due to frequent unforeseeable events. In this work we deal with process adaptability, i.e., the ability of the PMS to automatically cope with deviations, and we devise a sound and complete technique suitable for sequential processes running concurrently. The technique is based on a general framework which adopts the Situation Calculus, ConGolog and regression planning as basic elements. The applicability to the challenging scenario of emergency management is also shown.

Categories and Subject Descriptors

I.6.7 [**Software Engineering**]: Software/Program Verification—*Validation*; H.4.1 [**Information Systems Applications**]: Office Automation—*Workflow management*

General Terms

Verification

Keywords

Situation Calculus, GOLOG, Process Management, Pervasive Scenarios, Smart Devices

1. INTRODUCTION

Process Managements Systems (PMSs) are currently used mainly for handling the performance of traditional business scenarios, such as banks or insurance companies. Besides these settings, which present mainly static characteristics, PMSs can be used also in pervasive and highly dynamic situations, such as emergency situations, pervasive healthcare or domotics/home automation.

Let us consider, for example, a scenario for emergency management as introduced in [5]. There, a PMS can be used to coordinate the activities of emergency operators within teams. The members of a team are equipped with PDAs and are coordinated through the PMS residing on a leader device (usually an ultra-mobile laptop). In such a PMS, process schemas (in the form of Activity Diagrams) are defined, describing different aspects, such as tasks/activities, control and data flow, tasks assignment to services, etc. Every task is associated to a set of conditions which ought to be true for the task to be performed; conditions are defined on control and data flow (e.g., a previous task has to be completed or a variable needs to be assigned a specific range of values). Devices communicate among themselves through ad hoc networks [9]; and in order to carry on the whole process, such devices need to be continually connected to the PMS. However, this cannot be guaranteed: the environment is highly dynamic and the movement of nodes (that is, devices and related operators) within the affected area, while carrying out assigned tasks, can cause disconnections and, thus, unavailability of nodes. This means that in highly dynamic scenarios, processes can be easily invalidated since the execution environment may change continuously because of frequent unforeseeable events. In such cases, the process can no longer proceed. Therefore, some type of *process adaptability* is desirable in such scenarios. But what does "adaptability" mean? Adaptability can be seen as the ability of the PMS to reduce the gap from the *virtual 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 [3]. For instance in scenarios of emergency management, in virtual reality PMS assumes nodes to be always connected. But in physical reality when nodes are moving, they can lose a wireless connection and, hence, may be unable to communicate.

The reduction of this gap requires sufficient knowledge of both kinds of realities (virtual and physical). Such knowledge, harvested by the services performing the process's tasks, would allow the PMS to sense deviations and modify the process to ensure that its final goal is achieved. For instance, in order to handle the disconnection of a node X, the PMS might assign a task "Follow X" to another node Y in order to maintain the connection.

In classical PMSs applied to business scenarios, the procedure for handling possible run-time exceptions is generally subject to acknowledgement by the person responsible for the process. This authorization may be provided at run-time for handling deviations caused by a single exceptional event. Or, conversely, it is possible that the person gives the "go-ahead" for all exceptions in a certain class, defining how they should be handled. In any case, the adaptation is manual and requires human intervention. Conversely, here we are dealing with "mobile processes" in which participants are equipped with "mobile" devices and "move" in the environment to perform the tasks assigned. In these highly dynamic and pervasive scenarios, the execution environment changes continuously during the whole execution in unexpected ways. Deviations are frequent events and often, due to deadline constraints, they must often be handled very quickly. Such a requirement rules out waiting for a person's acknowledgement: adaptation must be as *automatic* and *autonomic* as possible.

The aims of this work are to improve upon what was proposed in [5], an approach based on planning techniques in AI. That approach synthesizes, if this is possible, a linear process (i.e., a process constituted only by a sequence of actions) which can recover the situation. This process is inserted at a given point of the original process – exactly the point in which the deviation was identified. In more details, let's assume that the current process is $\delta = (\delta_1; \delta_2)$ 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 where a deviation is identified. Then the technique aims to synthesize a linear process h that deals with the deviation. The adapted process is $\delta' = (\delta_1; h; \delta_2)$ and achieves every goal achieved also by δ

In this paper, we propose a novel adaptation technique that is more efficient, being able to exploit concurrent branches. In [5], whenever a process needs to be adapted, the different concurrently running branches are all interrupted. And a "repair" sequence of actions $h = [a_1, a_2, \ldots, a_n]$ is placed before them. Thus, all the branches can only resume execution after the "repair" sequence has been executed. This was done because one cannot adapt the branches individually without knowing whether the different branches act upon the same variables/conditions. Adapting branches one by one could cause other branches to be unable to progress.

Here we refine that approach by automatically identifying whether concurrent branches are independent (i.e., neither working on the same variables nor affecting some conditions). If independent, we can automatically synthesize a recovery process such that it affects only the interested branch, without having to block the other branches.

The rest of the paper is organized as follows: Section 2 introduces Situation Calculus and ConGolog which are the basis of our approach. Section 3 gives an overall idea of the adaptation approach, pointing out the general framework, whereas Section 4 presents the sound and complete technique for adapting "broken" processes. Section 5 outlines an example stemming from emergency management scenarios, showing the use of the proposed technique. Finally, Section 6 concludes the paper by highlighting the novelty wrt. the other research work and outlines future developments.

For the sake of brevity, we omit proofs; these can be found in [4].

2. PRELIMINARIES

In this paper, we use the situation calculus (SitCalc) to formalize adaptation in PMSs. The SitCalc is a logic formalism designed for representing and reasoning about dynamic domains [12]. 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. We abbreviate with $do([a_1, \ldots, a_{n-1}, a_n], s)$ the term $do(a_n, do(a_{n-1}, ..., do(a_1, s)))$, which denotes the situation obtained from s by performing the sequence of actions $a_1, \ldots, a_n$.

Conditions whose truth value may change are modeled by means of *fluents*, which are predicates that take a situation as their final argument. Fluents may be thought of as "properties" of the world whose values may vary across situations. We can write first order formulas using fluents. A formula is *uniform in situation s* if s is the only situation term that appears in it. Sometimes, we use *situation-suppressed* formulas; these are uniform formulas with all situation arguments suppressed (e.g. G denotes the situation-suppressed expression for $G(s)$).

Within the SitCalc, one can formulate action theories that formulate how the world changes as the result of the available actions. For example, basic action theories [12] include domain-independent foundational axioms that describe the structure of the situations, *(i)* initial state axioms that describe what is true initially and *(ii)* successor state axioms (one per fluent) specifying how fluents change as results of actions.[1]

The successor state axiom for a particular fluent F captures the effects and non-effect of actions on F and has the following form:

$$F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s) \qquad (1)$$

where $\Phi_F(\vec{x}, a, s)$ is a formula fully capturing the truth-value of fluent F on objects $\vec{x}$ when action a is performed in situation s ($\vec{x}$, a, and s are all the free variables in Φ_F).

On top of these theories of action, one can define complex control behaviors by means of high-level programs expressed in Golog-like programming languages [12]. In particular we focus on ConGolog [2], which is equipped with primitives for expressing concurrency. Table 1 summarizes the constructs of ConGolog used in this work. These constructs are sufficient for defining every well-structured process as defined in [7].

From a formal point of view, ConGolog programs are terms. The execution of ConGolog programs is expressed through the predicate $Do(\delta, s, s')$, which given the starting situation s and a program δ holds for all possible situations s' that result from executing δ starting from s. Notice that there may be more than one resulting situation since ConGolog programs can be non-deterministic (e.g., due to concurrency).

[1]For simplicity, we do not consider precondition axioms here; we assume that all actions are always possible.

Construct	Meaning
a	A primitive action
$\pi\vec{x}.\delta[\vec{x}]$	Nondeterministic choice of arguments $\vec{x}$
$\phi?$	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

Finally, our adaptation procedure will make use of *regression* (see [1] and [12]). Let $\varphi(do([a_1,\ldots,a_n],s))$ be a SitCalc formula with situation argument $do([a_1,\ldots,a_n],s)$. Then, $\mathcal{R}^s(\varphi(do([a_1,\ldots,a_n],s)))$ is the formula with situation s which denotes the facts/properties that must hold before executing $a_1,\ldots,a_n$ in situation s for $\varphi(do([a_1,\ldots,a_n],s))$ to hold (aka the weakest preconditions for obtaining φ). To compute the regressed formula $\mathcal{R}^s(\varphi(do([a_1,\ldots,a_n],s)))$ from $\varphi(do([a_1,\ldots,a_n],s))$, one iteratively replaces every occurrence of a fluent with the right-hand side of its successor state axiom 1 until every atomic formula has a situation argument that is simply s.

3. GENERAL FRAMEWORK

The general framework which we introduce in this paper is based on execution monitoring formally represented in the SitCalc [8, 3]. 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.

In this framework, every process is considered as a ConGolog program and every task t as a predefined sequence of two actions, which are executed atomically by the PMS:

$Start(w,t(x))$ task t is started by service w on input x

$Stop(w,t(y))$ task t is completed by service w returning output y.

In addition, there are two sets of actions $setPicked(w)$ and $unsetPicked(w)$ to update proper fluents to state that service w is assigned to a task t and, hence, it cannot be assigned to any other.

Finally, we define the concept of capability, which is used in binding tasks to services which are qualified to execute them. In order to do that, we assume that the process designers define the following situation-independent predicates:

$provide(w,c)$ service w provides capability c

$require(x,b)$ task x requires the capability c

Before a process starts execution, the PMS takes a set of facts denoting the initial context in the real environment as initial situation S_0, together with program (i.e., the process) δ_0 to be carried on, which the process designers devise in a phase preceding the actual execution. 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.

At each step, the 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 is represented as fluents that are defined on situations. The current state corresponds to the boolean values of these fluents evaluated on the current situation.

The process execution is continuously monitored to detect any deviation between physical reality and virtual reality. The PMS collects data from the environment through *sensors* (here *sensor* is any software or hardware component enabling to retrieve contextual information). If a deviation is sensed between the virtual reality as represented by s' and the physical reality as s'_e, the PMS internally generates a discrepancy $\vec{e} = (e_1, e_2, \ldots, e_n)$, which is a sequence of actions called exogenous events such that $s'_e = do(\vec{e}, s')$.[2]

Let us consider the case in which the remaining program-process to be executed δ is composed by n parallel sub-processes running concurrently: $\delta = [\vec{p}_1 \parallel \ldots \parallel \vec{p}_n]$ where every sub-process $\vec{p_i} = [a_{1,i},\ldots,a_{m,i}]$ is a sequence of simple actions.[3]

The process designers are assumed to have associated every sub-process p_i with the goal G_i that p_i is meant to achieve before the process enactment. In addition, the concurrent sub-processes are also annotated with an invariant condition C, expressed in the SitCalc. Independence of these sub-processes is maintained assuming this condition C, which must hold in the actual situation. Checking for independence is a key point of the adaptation technique proposed in this work (see next section).

After the divergence between the virtual and physical reality because of exogenous events, one or more concurrent processes can become broken (i.e, they no longer achieve the associated goals). For each broken branch p_i, the recovery procedure generates a handler h_i, which is an action sequence that, when placed before p_i, allows $p'_i = (h_i; p_i)$ to reach goal G_i and, while remaining independent of every parallel branch p_j (with $j \neq i$) with respect to invariant C.

4. ADAPTATION OF INDEPENDENT CONCURRENT PROCESSES

This section describes the approach we use to adapt a process composed of concurrent sequential sub-processes. We first give the formal foundations of our adaptation technique, presenting the results that the "monitor and repair" cycle relies upon. Then, we describe the "monitor and repair" cycle, and discuss the conditions under which the technique is sound and complete.

4.1 Formalization

In order to capture formally the concept of independence among processes, we introduce some preliminary definitions.

Definition 1. A ground action a preserves the achievement of goal G by a sequence of ground actions $[a_1,\ldots,a_n]$ under condition C if executing a at any point during $[a_1,\ldots,a_n]$ does not affect any of the conditions that are required for the goal G to be achieved by $[a_1,\ldots,a_n]$. More-

[2]Note that the action sequence $\vec{e}$ might not be the one that really occurred.

[3]If thsi assumption does not hold, the approach in [5] is still usable and we do not propose any improvement.

over, executing a preserves C in any situation. Formally:

$$\begin{aligned}
&PreserveAch(a, G, [a_1, \ldots, a_m], C) \overset{\text{def}}{=} \forall s.C(s) \Rightarrow \\
&\quad C(do(a, s)) \wedge \\
&\quad (G(do([a_1, \ldots, a_m], s)) \Rightarrow G(do([a, a_1, \ldots, a_m], s))) \wedge \\
&\quad (G(do([a_2, \ldots, a_m], s)) \Rightarrow G(do([a, a_2, \ldots, a_m], s))) \wedge \\
&\quad \ldots \\
&\quad (G(do(a_m, s)) \Rightarrow G(do([a, a_m], s))) \wedge \\
&\quad (G(s) \Rightarrow G(do(a, s))).
\end{aligned}$$

We then extend the notion above to the case of action sequences:

Definition 2. A sequence of ground actions $[a_1, \ldots, a_m]$ preserves the achievement of goal G by a sequence of ground actions $\vec{p}$ under condition C if every action in $[a_1, \ldots, a_m]$ preserves the achievement of goal G by $\vec{p}$ under condition C. Formally:

$$\begin{aligned}
&PreserveAch([a_1, \ldots, a_m], G, \vec{p}, C) \overset{\text{def}}{=} \\
&\quad \textstyle\bigwedge_{i:1 \le i \le n} PreserveAch(a_i, G, \vec{p}, C).
\end{aligned}$$

Given this, we can then define a notion of *independence of processes.*

Definition 3. A set of (sequential) processes $\vec{p}_1, \ldots, \vec{p}_n$ where each $\vec{p}_i$ achieves goal G_i are *independent* with respect to goals G_1 to G_n under condition C if for all i and all $j \neq i$, $\vec{p}_j$ preserves the achievement of goal G_i by $\vec{p}_i$ under condition C. Formally:

$$\begin{aligned}
&IndepProcess([\vec{p}_1, \ldots, \vec{p}_n], [G_1, \ldots . G_n], C) \overset{\text{def}}{=} \\
&\quad \textstyle\bigwedge_{i:1 \le i \le n} \bigwedge_{j:1 \le j \le n \wedge j \neq i} PreserveAch(\vec{p}_j, G_i, \vec{p}_i, C).
\end{aligned}$$

Basically, Definition 3 looks at the independence of each and every pair of concurrent (sub-)processes. If we assume that every process is composed by m actions, checking this independence is polynomial in the number of actions and concurrent processes. Specifically it requires

$$\binom{n}{2} \times m^2 = O(m^2 \times n^2) \qquad (2)$$

checks of $PreserveAch(\cdot)$ as in Definition 1 (one for each pair of actions in the concurrent processes).

The following theorem shows that if n processes $\vec{p}_1, \ldots, \vec{p}_n$ achieve their respective goals $G_1, \ldots, G_n$ and are independent according to Definition 3 with respect to a certain condition C, then any interleaving of the execution of the processes' actions will achieve each process's goal, and condition C will continue to hold. Let $\mathcal{D}$ the current domain theory. Then:

THEOREM 1.

$$\begin{aligned}
&\mathcal{D} \models IndepProcess([\vec{p}_1, \ldots, \vec{p}_n], [G_1, \ldots . G_n], C) \Rightarrow \\
&\quad \forall s.G_1(do([\vec{p}_1], s)) \wedge \ldots \wedge G_n(do([\vec{p}_n], s)) \wedge C(s) \Rightarrow \\
&\qquad [\forall s'.Do([\vec{p}_1 \parallel \ldots \parallel \vec{p}_n], s, s') \Rightarrow \\
&\qquad G_1(s') \wedge \ldots \wedge G_n(s') \wedge C(s')].
\end{aligned}$$

Next we show that if the concurrent sub-processes are independent and some of them progress, then the parts of them that remain to be executed will always remain independent:

THEOREM 2. *For each $i \le n$ and for all suffixes $\vec{p}_i'$ of $\vec{p}_i$*

$$\begin{aligned}
&\mathcal{D} \models IndepProcess([\vec{p}_1, \ldots, \vec{p}_n], [G_1, \ldots . G_n], C) \Rightarrow \\
&\qquad IndepProcess([\vec{p}_1', \ldots, \vec{p}_n'], [G_1, \ldots . G_n], C).
\end{aligned}$$

The theorem follows from the fact that the conditions required for independence of a set of processes include those required for independence of a set of suffixes of the processes.

Now we show that adding an action sequence $\vec{h}$ for handling a discrepancy $\vec{e}$, breaking a process, namely p_i, will preserve process independence, provided that $\vec{h}$ is built as independent of every sub-process different from $\vec{p}_i$ with respect to condition C. Let $\mathcal{R}(G_i, do(\vec{p}_i))$ be the situation-suppressed expression for regression $\mathcal{R}^s(G_i(do(\vec{p}_i, s))$.

THEOREM 3. *Let $\vec{p}_i'$ be the process broken by a discrepancy $\vec{e}$.*

$$\begin{aligned}
&\mathcal{D} \models IndepProcess([\vec{p}_1', \ldots, \vec{p}_{i-1}', \vec{p}_i', \vec{p}_{i+1}', \ldots, \vec{p}_n'], \\
&\quad [G_1, \ldots, G_{i-1}, G_i, G_{i+1}, \ldots, G_n], C) \wedge \\
&\quad \textstyle\bigwedge_{j:1 \le j \le n \wedge j \neq i} \big(PreserveAch(\vec{h}, G_j, \vec{p}_j', C) \\
&\quad \wedge PreserveAch(\vec{p}_j', \mathcal{R}(G_i, \vec{p}_i), \vec{h}, C\big) \Rightarrow \\
&\qquad IndepProcess([\vec{p}_1', \ldots, \vec{p}_{i-1}', [\vec{h}; \vec{p}_i'], \vec{p}_{i+1}', \ldots, \vec{p}_n'], \\
&\qquad [G_1, \ldots, G_{i-1}, G_i, G_{i+1}, \ldots, G_n], C).
\end{aligned}$$

The theorem stems from the application of Definition 3 concerning independence of processes. Finally, building on the previous results, we show that such an "independent handler" $\vec{h}$ can be used for handling a discrepancy $\vec{e}$ breaking a process p_i, while allowing all other processes to execute concurrently and achieve their respective goals:

THEOREM 4. *Let $\vec{p}_i'$ be the process broken by a discrepancy $\vec{e}$.*

$$\begin{aligned}
&\mathcal{D} \models \forall s, \vec{e}.IndepProcess([\vec{p}_1', \ldots, \vec{p}_{i-1}', [\vec{h}; \vec{p}_i'], \vec{p}_{i+1}', \ldots, \vec{p}_n'], \\
&\quad [G_1, \ldots, G_{i-1}, G_i, G_{i+1}, \ldots, G_n], C) \wedge \\
&\quad G_1(do([\vec{p}_1'], do(\vec{e}, s))) \wedge \ldots \wedge G_{i-1}(do([\vec{p}_{i-1}'], do(\vec{e}, s))) \wedge \\
&\quad G_{i+1}(do([\vec{p}_{i+1}'], do(\vec{e}, s))) \wedge \ldots \wedge G_n(do([\vec{p}_n'], do(\vec{e}, s))) \wedge \\
&\quad G_i(do([\vec{h}, \vec{p}_i'], do(\vec{e}, s))) \wedge C(do(\vec{e}, s)) \Rightarrow \\
&\qquad [\forall s'.Do([\vec{p}_1' \parallel \ldots \parallel \vec{p}_{i-1}' \parallel [\vec{h}, \vec{p}_i'] \parallel \vec{p}_{i+1}' \parallel \ldots \parallel \vec{p}_n'], \\
&\qquad do(\vec{e}, s), s') \Rightarrow G_1(s') \wedge \ldots \wedge G_{i-1}(s') \wedge G_i(s') \\
&\qquad \wedge G_{i+1}(s') \wedge \ldots \wedge G_n(s') \wedge C(s')].
\end{aligned}$$

4.2 Monitoring-Repairing Technique

On the basis of the results in the previous section, we propose in Figure 1 an algorithm for adaptation. This algorithm, which is meant to run inside the PMS, relies on 2 arrays giving information about the status of the n processes concurrently running: whether each is completed or not and, in case of completion, whether successfully or unsuccessfully. Initially every element of both arrays is set to `false`.

Routine MONITOR relies on every process p_i sending a message to the PMS when it either terminates successfully (message $successfullycompleted(i)$) or an exception is sensed such that such p_i can no longer terminate successfully[4] (message $exception(i_e, s_e)$ where i_e is the "broken" process and

[4]That is $\mathcal{D} \not\models \mathcal{R}^{s_{now}}(G_i(do(p_i, s_{now})))$ where s_{now} is the current situation after sensing a discrepancy.

```
completed[]: array of n elements
succeeded[]: array of n elements
INITIALLY()
 1  for $i \leftarrow 1$ to $n$
 2  do $completed[i] \leftarrow$ false
 3      $succeeded[i] \leftarrow$ false

SUB MONITOR($[p_1, \ldots, p_n], [G_1, \ldots, G_N], C, s_i$)
 1  if $(\neg IndepProcess([p_1, \ldots, p_n], [G_1, \ldots, G_N], C))$
 2     then throw exception
 3  while $(\exists i. \neg completed[i])$
 4  do $m \leftarrow$ WAITFORMESSAGE()
 5      if $m = successfullyCompleted(i)$
 6         then $completed[i] \leftarrow$ true
 7              $succeeded[i] \leftarrow$ true
 8      if $m = exception(i_e, s_e)$
 9         then $h \leftarrow$ BUILDHANDLER($i_e, s_e, [p_1, \ldots, p_n], [G_1, \ldots, G_n], C$)
10              if $h =$ fail
11                 then $completed[i_e] \leftarrow$ true
12                      throw exception
13                 else
14                      $p_{i_e} \leftarrow [h; p_{i_e}]$
15                      START($p_{i_e}$)

FUNCTION BUILDHANDLER($i_e, s_e, [p_1, \ldots, p_n], [G_1, \ldots, G_n], C$)
 1  for $i \leftarrow 1$ to $n$
 2  do $p_i \leftarrow remains(p_i, s_e)$
 3  $h \leftarrow$ PLANBYREGRES($\mathcal{R}(G_{i_e}, do(\vec{p}_{i_e}))$),
 4       $s_e, [p_1, \ldots, p_n]/p_{i_e}, [G_1, \ldots, G_n]/G_{i_e}, C$)

FUNCTION PLANBYREGRES($Goal_h, s_i, [p_1, \ldots, p_{n-1}], [G_1, \ldots, G_{n-1}], C$)
 1  if $\mathcal{D} \models Goal_h(s_i)$
 2     then return nil
 3     else  $a \leftarrow$ CHOOSEACTION($Goal_h, s_i, [p_1, \ldots, p_{n-1}]$,
 4                 $[G_1, \ldots, G_{n-1}], C$)
 5           if $a =$ nil
 6              then
 7                   return fail
 8              else
 9                   $h' \leftarrow$ PLANBYREGRES($\mathcal{R}^s(Goal_h(do(a, s)))$,
10                        $s_i, [p_1, \ldots, p_{n-1}], [G_1, \ldots, G_{n-1}], C$)
11                   if $h =$ fail
12                      then return fail
13                      else  return $[h'; a]$

FUNCTION CHOOSEACTION($Goal_h, s_i, [p_1, \ldots, p_{n-1}], [G_1, \ldots, G_N], C$)
 1  choose an action $a$ s.t. $\{\exists s(\neg Goal_h(s) \wedge Goal_h(do(a, s)))\} \wedge$
 2     $\forall i. 1 \leq i \leq (n-1) \Rightarrow PreserveAch(a, G_i, p_i, C)$
 3     $\wedge\ PreserveAch(p_i, Goal_h, a, C)$
 4  return $a$ //even nil if there exists no selectable action
```

Figure 1: Pseudo-Code for the adaptation technique.

s_e is the resulting situation after the discrepancy occurrence). We assume that the situation representing the current state in the real word is known and that we have complete knowledge of the fluents in that situation. Moreover we assume that in every situation we can get access to the fluent values in every past situation.[5]

The routine is applicable if all processes are independent of each other. Therefore, before starting its monitoring and repairing, it checks whether the process independence assumption holds (lines 1,2). If not, it throws an exception, assuming that in this case an alternative and more intrusive approach would be used [5].

Later on, in the "monitor and repair" cycle, we listen for arriving messages (line 4). If the message concerns the successful completion of a sub-process, then the arrays are updated accordingly (lines 5-7). Otherwise, the message is about a sub-process p_{i_e} that has been broken by a discrepancy. p_{i_e} is implicitly halted and we call function BUILDHANDLER to search for an adaptation handler h. If such a handler h is found, it is prefixed to the broken process p_{i_e}, which becomes $(h; p_{i_e})$ (line 14). Finally, the adapted process is started again (line 15).

How does the BUILDHANDLER function synthesize this handler? Lines 1-2 update all processes p_i so that they represent the subparts that remain to execute. Then, the function invokes a regression planner (line 3) [11, 6], which searches for a plan backwards from the goal description. Specifically, the regression planner tries to generate a sequential plan that, starting from the current situation s_e, arrives at some situation s_h such that p_{i_e} can be executed again and achieve G_{i_e}, i.e. $\mathcal{R}^{s_h}(G_i(do(p_{i_e}, s_h)))$.

The regression planning procedure PLANBYREGRESSION recursively and incrementally builds a plan[6] checking that every selected action is independent of each p_j (with $j \neq i_e$) with respect to invariant condition C. Indeed, Theorems 3 and 4 ensure that if the handler only includes actions that are independent of each p_j (with $j \neq i_e$), then, for all possible interleavings, process $(h; p_{i_e})$ will achieve its goal G_i and every other process p_j with $j \neq i_e$ will continue to achieve its goals G_j.

Observe that Theorem 2 ensures if processes were originally independent regarding their respective goals and no exceptions are raised, as they evolve, they remain independent.

Next we focus on the soundness and completeness of the algorithm proposed.

Let $\vec{e}$ be a discrepancy which breaks one process, namely p_i, and let $\bar{s}$ be the situation before the discrepancy occurrence. Then:

THEOREM 5 (SOUNDNESS). *If the algorithm in Figure 1 produces an handler:*

$$\vec{h}_1 = \text{BUILDHANDLER}(i, do(\vec{e}, \bar{s}), [p_1, \ldots, p_n], [G_1, \ldots, G_n], C)$$

and

$$IndepProcess([p_1, \ldots, p_n], [G_1, \ldots, G_n], C) \wedge G_1(do(p_1, \bar{s})) \wedge \ldots \wedge G_n(do(p_n, \bar{s}))$$

then

$$\forall s'. Do([\vec{p}_1 \parallel \ldots \parallel \vec{p}_{i-1} \parallel (\vec{h}_i; \vec{p}_i) \parallel \vec{p}_{i+1} \parallel \ldots \parallel \vec{p}_n], do(\vec{e}, \bar{s}), s') \Rightarrow G_1(s') \wedge G_n(s') \wedge C(s')]$$

and

$$IndepProcess([\vec{p}_1 \parallel \ldots \parallel \vec{p}_{i-1} \parallel (\vec{h}_i; \vec{p}_i) \parallel \vec{p}_{i+1} \parallel \ldots \parallel \vec{p}_n], [G_1, \ldots, G_n], C)$$

THEOREM 6 (COMPLETENESS). *If*

$$\exists \underline{\vec{h}}. (\forall s'. Do([\vec{p}_1 \parallel \ldots \parallel \vec{p}_{i-1} \parallel (\underline{\vec{h}}; \vec{p}_i) \parallel \vec{p}_{i+1} \parallel \ldots \parallel \vec{p}_n], do(\vec{e}, \bar{s}), s') \Rightarrow G_1(s') \wedge \ldots \wedge G_n(s') \wedge C(s')) \wedge IndepProcess([p_1, \ldots, p_{i-1}(\underline{\vec{h}}; p_i), , p_{i+1}, \ldots, p_n], [G_1, \ldots, G_n], C)$$

then BUILDHANDLER *returns a repairing handler:*

$$\vec{h} = \text{BUILDHANDLER}(i, do(\vec{e}, \bar{s}), [p_1, \ldots, p_n], [G_1, \ldots, G_n], C)$$

[5]This could be done by logging and storing them in a repository.

[6]Here we assume that plans are returned in form of CONGOLOG programs.

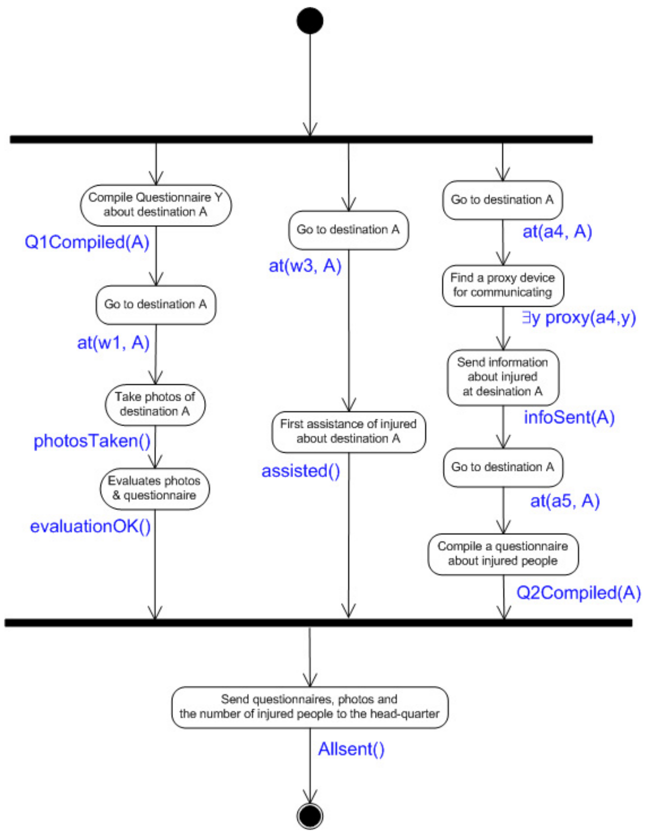
Compile Questionnaire Y about destination A
Q1Compiled(A)
Go to destination A
at(w1, A)
Take photos of destination A
photosTaken()
Evaluates photos & questionnaire
evaluationOK()
Go to destination A
at(w3, A)
First assistance of injured about destination A
assisted()
Go to destination A
at(a4, A)
Find a proxy device for communicating
∃y proxy(a4,y)
Send information about injured at desination A
infoSent(A)
Go to destination A
at(a5, A)
Compile a questionnaire about injured people
Q2Compiled(A)
Send questionnaires, photos and the number of injured people to the head-quarter
Allsent()

Figure 2: A possible process to be carried on in disaster management scenarios

such that

$$\begin{aligned}&\forall s'.Do([\vec{p}_1 \parallel \ldots \parallel \vec{p}_{i-1} \parallel (\vec{h};\vec{p}_i) \parallel \vec{p}_{i+1} \parallel \ldots \parallel \vec{p}_n],\\ &\quad do(\vec{e},\bar{s}), s') \Rightarrow G_1(s') \wedge \ldots \wedge G_n(s') \wedge C(s')\\ &\wedge\ IndepProcess([p_1,\ldots,(h;p_i),p_n],[G_1,\ldots,G_i,\ldots,G_n],C)\end{aligned}$$

Note that the above procedure becomes easily realizable in practice if the PMS works in a finite domain (e.g., using discretized positions based on actual GPS positions) and propositional logic is sufficient. In fact, one can use an off-the-shelf regression planner such those mentioned in [11, 6].

5. AN EXAMPLE FROM EMERGENCY MANAGEMENT

In this section, we discuss an example of adaptation involving emergency management. 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 for all scenarios where one needs 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.

Figure 2 shows a (slightly simplified) example of a possible scenario for the aftermath of an earthquake. Some actors are assessing the area for dangerous partially-collapsed buildings. Meanwhile others are giving first aid to the injured people and sending information about required ambulances and filling in a questionnaire about the injured people, which are required by the headquarter. The corresponding ConGolog program is depicted in Figure 3.

In order to formalize the scenario in Figure 2 we rely on two fluents independent of any specific domain:

$started(w,t,s)$ is true in situation s, if task t has been started by service w and not yet stopped. For all task t in starting situation S_0, we have $started(t,S_0) = \texttt{false}$

$busy(w,s)$ is true in situation s, if service w has been assigned to a task. For all service w we have $busy(w,S_0) = \texttt{false}$.

Their successor-state axioms are:

$$\begin{aligned}&busy(w,do(x,s)) \Leftrightarrow \neg busy(w,s) \wedge x = setPicked(w) \vee\\ &\quad busy(w,s) \wedge x \neq unsetPicked(w)\\ &started(w,t,do(x,s)) \Leftrightarrow \forall z.\neg started(z,t,s) \wedge\\ &\quad \exists o.x = Start(w,t(o)) \vee started(w,t,s)\\ &\quad \wedge \forall i.x \neq Stop(w,t(i)).\end{aligned}$$

In addition, we use some domain-specific fluents. In the activity diagram in Figure 2, we have labeled every task with the fluents (in situation-suppressed form) that become true after the task's execution. For these fluents to become true (i.e., the respective tasks to achieve their post-conditions), it is ncessary that the respective tasks be correctly started and stopped through the proper actions. For sake of brevity, we discuss only those fluents that we are going to use in the example (see later in this section for the successor state axioms):

$proxy(w,y,s)$ is true if in situation s, service y can work as proxy for service w. In the starting situation S_0 for every pair of services w, y we have $proxy(w,y,S_0) = \texttt{false}$, denoting that no proxy has yet been chosen for w.

$at(w,p,s)$ is true if service w is located at coordinate $p = \langle p_x, p_y, p_z \rangle$ in situation s. In the starting situation S_0, for each service w_i, we have $at(w_i,p_i,S_0)$ where location p_i is obtained through GPS sensors.

$infoSent(d,s)$ is true in situation s if the information concerning injured people at destination d has been successfully forwarded to the headquarter. For all destinations d $infoSent(d,S_0) = \texttt{false}$.

$evaluationOK(s)$ is true if the photos taken are judged as having a good quality, with $evaluationOK(S_0) = \texttt{false}$.

```
MAIN()
  $\pi.w_0[available(w_0) \land \forall c.require(c, \texttt{QuestBuildings})$
    $\Rightarrow provide(w_0, c)]$;
      $setPicked(w_0)$;
  $\pi.w_1[available(w_1) \land \forall c.require(c, \texttt{TakePhoto})$
    $\Rightarrow provide(w_1, c)]$;
      $setPicked(w_1)$;
  $\pi.w_2[available(w_2) \land \forall c.require(c, \texttt{EvalPhoto})$
    $\Rightarrow provide(w_2, c)]$;
      $setPicked(w_2)$;
  $\pi.w_3[available(w_3) \land \forall c.require(c, \texttt{FirstAid})$
    $\Rightarrow provide(w_3, c)]$;
      $setPicked(w_3)$;
  $\pi.w_4[available(w_4) \land \forall c.require(c, \texttt{InformInjured})$
    $\Rightarrow provide(w_4, c)]$;
      $setPicked(w_4)$;
  $\pi.w_5[available(w_5) \land \forall c.require(c, \texttt{InjuredQuest})$
    $\Rightarrow provide(w_5, c)]$;
      $setPicked(w_5)$;
  (EVALTAKE($w_0, w_1, w_2, Loc, Q_1, F$) || ASSISTINJURED($w_3, Loc$) ||
    REPORTASSISTANCEINJURED($w_4, w_5, Loc, Q_2$));
  $unsetPicked(w_0)$; $unsetPicked(w_1)$; $unsetPicked(w_2)$;
  $unsetPicked(w_3)$; $unsetPicked(w_4)$; $unsetPicked(w_5)$;
  $\pi.w_6[available(w_6) \land \forall c.require(c, \texttt{SendByGPRS})$
    $\Rightarrow provide(w_6, c)]$;
      $setPicked(w_6)$;
  $Start(w_6, \texttt{SendByGPRS}(\{Q_1, F, Q_2\}))$;
  $Stop(w_6, \texttt{SendByGPRS}(\{\}))$;

EVALTAKE($w_0, w_1, w_2, Loc, Q_1, F$)
  $Start(w_0, \texttt{QuestBuildings}(\{Loc\}))$;
  $Stop(w_0, \texttt{QuestBuildings}(\{Q_1\}))$;
  $Start(w_1, \texttt{Go}(\{Loc\}))$;
  $Stop(w_1, \texttt{Go}(\{LocGone\}))$;
  $Start(w_1, \texttt{TakePhoto}(\{Loc\}))$;
  $Stop(w_1, \texttt{TakePhoto}(\{F\}))$;
  $Start(w_2, \texttt{Evaluate}(\{Loc, Q_1, F\}))$;
  $Stop(w_2, \texttt{Evaluate}(\{o\}))$;

ASSISTINJURED($w_3, Loc$)
  $Start(w_3, \texttt{Go}(\{Loc\}))$;
  $Stop(w_3, \texttt{Go}(\{LocGone\}))$;
  $Start(w_3, \texttt{FirstAid}(\{Loc\}))$;
  $Stop(w_3, \texttt{FirstAid}())$;

REPORTASSISTANCEINJURED($w_4, w_5, Loc, Q_2$)
  $Start(w_4, \texttt{Go}(\{Loc\}))$;
  $Stop(w_4, \texttt{Go}(\{LocGone\}))$;
  $Start(w_4, \texttt{FindProxy}(\{Loc\}))$;
  $Stop(w_4, \texttt{FindProxy}())$;
  $Start(w_4, \texttt{InformInjured}(\{Loc\}))$;
  $Stop(w_4, \texttt{InformInjured}())$;
  $Start(w_5, \texttt{Go}(\{Loc\}))$;
  $Stop(w_5, \texttt{Go}(\{LocGone\}))$;
  $Start(w_5, \texttt{InjuredQuest}(\{Loc\}))$;
  $Stop(w_5, \texttt{InjuredQuest}(\{Q_2\}))$;
```

Figure 3: The ConGolog program corresponding to the process in Figure 2

$assisted(z, s)$ is true if the injured people in area z have been supported through a first-aid medical assistance. We have $assisted(S_0) = \texttt{false}$.

For this example, we assume that the process designers have defined the goals of the three concurrent sub-processes as follow:

$$G_1(s) \overset{\text{def}}{=} Q1Compiled(A, s) \land EvaluationOK(s)$$
$$G_2(s) \overset{\text{def}}{=} assisted(A, s)$$
$$G_3(s) \overset{\text{def}}{=} Q2Compiled(A, s) \land infoSent(A, s)$$

In addition, we are using in this example the invariant condition $C(s) = \texttt{true}$ for all situations s, meaning that we are not using any assumption to show process independence.

Before formally specifying the aforementioned fluents, we define some abbreviations:

$available(w, s)$: which states a service w is available if it is connected to the coordinator device (denoted by $\texttt{Coord}$) and is free.

$connected(w, z, s)$: which is true if in situation s the services w and z are connected through possibly multi-hop paths.

$neigh(w, z, s)$: which holds if the services w and z are in radio-range in the situation s.

Their definitions are as follows:

$$\begin{aligned}
&neigh(w_0, w_1, s) \overset{\text{def}}{=} at(w_0, p_0, s) \land at(w_1, p_1, s) \\
&\quad \land \parallel p_0 - p_1 \parallel < rrange \\
&connected(w_0, w_1, s) \overset{\text{def}}{=} neigh(w_0, w_1, s) \\
&\quad \lor \exists w_2.neigh(w_0, w_2, s) \\
&\qquad \land neigh(w_2, w_1, s) \\
&\quad \lor \exists w_2, w_3.neigh(w_0, w_2, s) \\
&\qquad \land neigh(w_2, w_3, s) \land neigh(w_3, w_1, s) \\
&\quad \lor \exists w_2, w_3, w_4 \ldots \\
&\quad \lor \ldots \lor \exists w_2, w_3, \ldots, w_n neigh(w_0, w_2, s) \\
&\qquad \land neigh(w_2, w_3, s) \land neigh(w_3, w_1, s) \land \ldots \\
&available(w, s) \overset{\text{def}}{=} \neg busy(w, s) \land connected(w, \texttt{Coord}, s))
\end{aligned}$$

In the above, n is the actual number of services.

The successor state axioms for the aforementioned fluents are as follows:

$$\begin{aligned}
&at(w, p, do(x, s)) \Leftrightarrow \\
&\quad \big((\forall\ p, q.\ x \neq Stop(w, \texttt{Go}(q))) \land at(w, p, s)\big) \lor \\
&\qquad x = Stop(w, \texttt{Go}(p)) \land started(w, \texttt{Go}, s) \\
&proxy(w, y, do(x, s)) \Leftrightarrow \\
&\quad \exists w.\ \big(x = Stop(w, \texttt{FindProxy}()) \land \\
&\qquad started(w, \texttt{FindProxy}, s) \land isClosestAvail(w, y, s)) \\
&\quad \lor \big(proxy(w, y, s) \land \forall w.\ x \neq Stop(w, \texttt{FindProxy}())\big) \\
&infoSent(d, do(x, s)) \Leftrightarrow \\
&\quad infoSent(w, s) \lor \neg infoSent(w, s) \land \\
&\quad \exists w.(x = Stop(w, \texttt{InformInjured}()) \land at(w, d, s) \land \\
&\qquad started(w, \texttt{InformInjured}, s) \land \\
&\qquad \exists y.(proxy(w, y, z) \land neigh(w, y, s))\big) \\
&evaluationOK(do(x, s)) \Leftrightarrow \\
&\quad evaluationOK(s) \lor \big(\neg evaluationOK(s) \land \\
&\quad \exists w.(x = Stop(w, \texttt{EvalPhoto}(o)) \land o = isOk \land \\
&\qquad photosTaken(s) \land started(w, \texttt{EvalPhoto}, s))\big) \\
&assisted(z, do(x, s)) \Leftrightarrow \\
&\quad assisted(z, s) \lor big(\neg assisted(z, s) \land \\
&\quad \exists w.(x = Stop(w, \texttt{FirstAid}()) \land at(w, z, s) \land \\
&\qquad started(w, \texttt{FirstAid}, s))\big)
\end{aligned}$$

In the above, we use the abbreviation $isClosestAvail(w, y, s)$, which holds if y is the geographically closest service to w that is able to act as proxy; if there is no available proxy in w's radio range, $y = \mathsf{nil}$:

$$\begin{aligned}
&isClosestAvail(w, y, s) \overset{\text{def}}{=} \big(available(y) \land at(w, p_w, s) \land \\
&\quad at(y, p_y, s) \land provide(y, \texttt{proxy}) \land neigh(w, y, s) \\
&\quad \land \forall z.(z \neq y \land available(z) \land provide(z, \texttt{proxy}) \Rightarrow \\
&\qquad \parallel p_z - p_w \parallel > \parallel p_y - p_w \parallel)\big) \\
&\quad \lor \big(y = \mathsf{nil} \land \neg \exists z.\big(available(z) \land provide(z, \texttt{proxy}) \\
&\qquad \land neigh(w, z, s)\big)\big)
\end{aligned}$$

Note that we have not defined any rule to handle discrepancies. To show how our automatic adaptation technique is meant to work, let us consider an example of discrepancy and a handler plan to cope with it.

Firstly, let us consider the case where the process execution reaches line 6 of procedure ReportAssistanceInjured. At this stage, a proxy has been found, namely w_{pr} and the information about injured people is about to be sent to the headquarter to request a sufficient number of ambulances to take them to a hospital. Let $\bar{s}$ be the current situation. Of course, w_{pr} is selected to be in radio range of actor w_4: $neigh(w_4, w_{pr}, \bar{s})$.

Let's assume now that w_{pr} moves away for any reason to a position p'. This corresponds to a discrepancy $\vec{e} = [e_1; e_2]$ with $e_1 = Start(w_{pr}, \texttt{Go}(p'))$ and $e_2 = Stop(w_{pr}, \texttt{Go}(p'))$. So the new current situation is $s_e = do(\vec{e}, \bar{s})$ where $neigh(w_4, w_{pr}, \bar{s_e}) = \texttt{false}$. Consequently, action $Stop(w_{pr}, \texttt{InformInjured}())$ does not make $infoSent(A)$ become true as it was supposed to.

Since sub-processes EvalTake, AssistInjured and ReportAssistanceInjured are independent, the latter, which is affected by the discrepancy, can be repaired without having to stop the other processes.

The goal given to the regression planner is $Goal_h = photoTaken() \wedge infoSent(A) \wedge Q1Compiled(A)$ in situation-suppressed form. The following plan achieves this goal while preserving independence:

$$\begin{array}{l} Start(w_{pr}, \texttt{Go}(\{A\})); \\ Stop(w_{pr}, \texttt{Go}(\{A\})); \\ Start(w_4, \texttt{InformInjured}(\{A\})); \\ Stop(w_4, \texttt{InformInjured}()); \end{array}$$

Adaptation can be performed by inserting it after line 5 of procedure ReportAssistanceInjured, ensuring that it can achieve its goal without interfering with the other sub-processes.

6. CONCLUDING REMARKS

In this paper we have proposed a sound and complete technique for adapting sequential processes running concurrently. Such a technique improves, under the assumption of independence of the different processes, the one proposed by de Leoni et al. in [5], while adopting the same general framework based on planning techniques in AI.

In [5], whenever a process needs to be adapted, the different concurrently running branches are all interrupted. And a sequence of actions $h = [a_1, a_2, \ldots, a_n]$ is placed before them. Therefore, all of the branches could only resume after the execution of the whole sequence. The adaption technique proposed here works on identifying whether concurrent branches are independent (i.e., neither working on the same variables nor affecting some conditions). And, if independent, it can synthesize a recovery process that affects only the branch of interest, without having to block the other branches.

As in [5, 10], the approach proposed is not based on the idea of capturing expected exceptions, as most current approaches do, defining the behaviors triggered when special events occur. Conversely, our technique models (a subset of) the running environment and the actions' effects, without considering possible special exceptional events.

Note that the proposed technique is made possible by annotating processes in a "declarative" way. We assume that the process designer can annotate actions/sequences with the goals they are intended to achieve, and on the basis of such declared goals, independence among branches can be verified, and then a recovery process which affects only the branch of interest, without side-effects on the others, is synthesized.

We are currently developing a running prototype that exploits the technique proposed here by using the IndiGolog module developed by the Cognitive Robotics Group of the University of Toronto and state-of-the-art planners in the AI literature.

7. REFERENCES

[1] J. Baier and S. McIlraith. On planning with programs that sense. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR06)*, pages 492–502, Lake District, UK, June 2006.

[2] G. De Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog, A Concurrent Programming Language based on the Situation Calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[3] G. De Giacomo, R. Reiter, and M. Soutchanski. Execution Monitoring of High-Level Robot Programs. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98*, pages 453–465, 1998.

[4] M. de Leoni. *Adaptive Process Management in Pervasive and Highly Dynamic Scenarios*. PhD thesis, SAPIENZA - University of Rome, 2009.

[5] M. de Leoni, M. Mecella, and G. De Giacomo. Highly Dynamic Adaptation in Process Management Systems Through Execution Monitoring. In *In Proceedings of Business Process Management, 5th International Conference, BPM 2007*, pages 182–197, 2007.

[6] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[7] B. Kiepuszewski, A. H. ter Hofstede, and C. Bussler. On structured workflow modelling. In *Proceedings of Advanced Information Systems Engineering, 12th International Conference CAiSE 2000*, pages 431–445, 2000.

[8] Y. Lesperance and H.-K. Ng. Integrating Planning into Reactive High-level Robot Programs. In *Proceedings of the 2nd International Cognitive Robotics Workshop*, 2000.

[9] B. S. Manoj and A. Hubenko Baker. Communication Challenges in Emergency Response. *Communincation of ACM*, 50(3):51–53, 2007.

[10] M. Mecella. Adaptive Process Management. Issues and (Some) Solutions. In *Proceedings of the Third IEEE Workshop on Agile Cooperative Process-Aware Information Systems (ProGility 2008) @ IEEE WETICE'08*, 2008.

[11] J. L. Pollock. The logical foundations of goal-regression planning in autonomous agents. *Artificial Intelligence*, 106(2):267–334, 1998.

[12] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems.* MIT Press, 2001.