DIPARTIMENTO DI INFORMATICA
E SISTEMISTICA ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

SmartPM - Featuring Automatic Adaptation to Unplanned Exceptions

Massimiliano de Leoni
Andrea Marrella
Massimo Mecella
Sebastian Sardina

Technical Report n. 4, 2011

SmartPM – Featuring Automatic Adaptation to Unplanned Exceptions

Massimiliano de Leoni, Andrea Marrella, Massimo Mecella
SAPIENZA - Università di Roma, Rome, Italy
{deleoni,marrella,mecella}@dis.uniroma1.it
Sebastian Sardina
RMIT University, Melbourne, Australia
{sebastian.sardina}@rmit.edu.au

Abstract. Process Management Systems (PMSs) are currently more and more used as a supporting tool for cooperative processes in pervasive and highly dynamic situations, such as emergency situations or pervasive health-care. In these scenarios, the environment may change in a way that was not expected so as to prevent processes from being successfully carried out. The frequency and the types of unexpected changes are significantly higher and larger than in classical business domains. Therefore, a manual adaptation is not feasible as well as defining in advance how to manage all possible changes which may occur. This paper illustrates SmartPM, a model and a prototype PMS that features a set of sound and complete techniques to automatically cope with unplanned changes. Specifically, the theoretical foundation and the general framework will be described, as well as the concrete implementation and its validation.

Index Terms. Process Management, Automatic Adaptation, Emergency Management, Situation Calculus, IndiGolog

1. INTRODUCTION

Nowadays organizations are always trying to improve the performance of the processes they are part of. It does not matter whether such organizations are dealing with classical static business domains, such as loans, bank accounts or insurances, or with pervasive and highly dynamic scenarios. The demands are always the same: seeking more efficiency for their processes to reduce the time and the cost of their executions.

A Process Management System (PMS) [Weske 2007] is aimed at increasing the efficiency and effectiveness in the execution of processes. The core of a PMS is the engine that manages the process routing and decides which tasks are enabled for execution, by taking into account the control flow, the value of variables and other aspects. Once a task is ready for being assigned, the engine is also in charge of assigning it to proper participants; this step is performed by considering the participant "skills" required by the single task: a task will be assigned to that participant that provides all of the skills required. Participants are provided with a client application, part of the PMS, named *Task Handler* or Task-list Handler. It is aimed at receiving notifications of task assignments. Participants can, then, use this application to pick the next task to work on.

Nowadays, PMSs are widely used for the management of "administrative" processes characterized by clear and well-defined structures where contingencies are quite infrequent. Conversely, this paper turns its attention to highly dynamic and pervasive scenarios, such as emergency management or health care. They are characterized by being very dynamic, turbulent, and subject to higher frequency of unexpected contingencies than classical scenarios. Therefore, PMSs for pervasive scenarios should provide a higher degree of operational flexibility/adaptability.

The current-day leading commercial PMS products and research prototypes provide some techniques to react to exceptions and adapt process instances to mitigate their effects. As already described in [Eder and Liebhart 1995], exceptions can be classified as expected (i.e., foreseeable) and unexpected (i.e., unforeseeable), where expected exceptions can be modeled at design time. Modeling expected exceptions fit adequately when the number of possible ones is relatively small and they occur quite rarely. Such techniques can be classified in two categories:

—Manual adaptation of processes upon occurrences of discrepancies: after an exception, domain experts are in charge of changing the schema of the affected processes, which, otherwise, could not be carried out successfully. This approach is not always applicable in highly dynamic and pervasive scenarios, as human intervention may imply unacceptable delays.

—Automatic pre-planned adaptation: process schemas are meant to include the specific actions to cope with potential failures. For each class of exogenous events that are envisioned to occur, a specific contingency plan is defined a priori.

For unexpected exceptions, besides the manual adaptation, which is still applicable, automatic "unplanned" adaptation can be envisioned: process schemas are defined as if exogenous events could never occur; there is a monitor which is continuously looking for the occurrence of exogenous events. When some of them occur, the process is automatically adapted to mitigate the effects. The difference with

the pre-planned adaptation consists in that there exist no pre-planned policies, but the policy is built on the fly ad-hoc for the specific occurrence.

As widely discussed in Section 2, most of currently available PMSs provide mechanisms based on pre-planned adaptation. For simple and mainly static processes, this is feasible and valuable. However, in mobile and highly dynamic scenarios, there could be so many different potential exceptions that it is impossible to predefine how to recovery from all deviations. What is more, many exceptions may not even be foreseeable a priori.

This paper focuses on describing SmartPM (Smart Process Management) , a model and a proof-of-concept PMS (engine) featuring a set of techniques to improve the degree of *automatic* adaptation. Such techniques are able to automatically adapt processes without explicitly defining handlers/policies to recover from exogenous events and without the intervention of domain experts. To that end, we use a specialized version of the concept of adaptation from the field of agent-oriented programming [De Giacomo et al. 1998]. Specifically, adaptation in SmartPM is seen as reducing the gap between 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. Exogenous events may deviate the virtual reality from the physical reality. Clearly, the reduction of their gap requires sufficient knowledge of both kinds of realities. Such knowledge—harvested by specific sensors—would allow the PMS to sense deviations and to deal with their mitigation.

The techniques presented here are based on the Situation Calculus [Reiter 2001] logical framework and on automated planning techniques [Ghallab et al. 2004]. The environment, services and tasks are given in domain theories described in Situation Calculus and processes are IndiGolog [De Giacomo et al. 2009] programs. IndiGolog is a logical language that allows for defining programs with cycles, concurrency, conditional branching that rely on program steps that are actions of some domain theory expressed in Situation Calculus. An execution monitor is responsible for detecting whether the gap between the virtual and physical realities is such that some processes cannot terminate successfully. In that case, SmartPM provides mechanisms for adapting the process schemas that require no human intervention and no pre-defined handlers for specific exceptions. SmartPM adopts a *service-based* approach to process management (cf. [Weske 2007]), that is, tasks are executed by *services*. Even human actors are seen as services; appropriate applications, namely the aforementioned Task Handlers, mediate between human actors and the system in order to make such actors appear as services.

As a running example, we will consider a scenario for emergency management where processes show a complexity that is comparable to business settings. This scenario stems from the European research project WORKPAD[1], which has inspired the research presented in this paper and has provided its validation context. In such a scenario, the members of a team are equipped with PDAs and coordinated through a PMS residing on a leader device (usually a ruggedized netbook). Devices communicate with each other through ad-hoc networks; in order to carry on the

[1]Cf. `http://www.workpad-project.eu` and `http://cordis.europa.eu/ictresults/index.cfm?section=news&tpl=article&BrowsingType=Features&ID=91287&highlights=workpad`.

overall process, they need to be continually inter-connected. In the virtual reality, devices are supposed to be continuously connected. In the physical reality, though, the movement of nodes (i.e., devices and related operators) within the affected area may cause disconnections, thus making the two realities deviate. Disconnections result in the unavailability of nodes and their corresponding services. Now, from the collection of actual user requirements [de Leoni et al. 2007; Humayoun et al. 2009], it turns out that typical teams are formed by a few nodes (less than 10 units), and therefore a simple task reassignment is generally not feasible: there may not be two "similar" services available to perform a given task. In this context, the adaptation to recover from the disconnection of a node X may require the assignment of a task of the form "Follow X" to another node Y[2]. Please note that the recovery plan "Follow X" was not designed beforehand; it was built on the fly to deal with the disconnection. In the near future, a similar situation could be managed in a complete different way because of a different environmental state.

This paper extends previous work reported in [de Leoni et al. 2007; de Leoni et al. 2008; de Leoni 2009] (cited here in chronological order). In [de Leoni et al. 2007], an initial theoretical framework has been proposed without discussing possible proof-of-concept implementations. Here, the theoretical framework has been refined and extended to be actually feasible in practice. Moreover, the framework has been extended to enhance its expressiveness. Paper [de Leoni et al. 2008] has illustrated a first framework implementation in which the adaptation technique was not yet present. Finally, with respect to [de Leoni 2009], this paper illustrates *(i)* the proof-of-concept implementation that allows for more expressive process models and includes adaptability; *(ii)* a technique to map WS-BPEL process models to SmartPM models; *(iii)* the validation in order to show the practical feasibility of the adaptation approach. With respect to previous works related to the WORKPAD project [Catarci et al. 2006; Catarci et al. 2007; Catarci et al. 2008; Battista et al. 2008; Humayoun et al. 2009; Mecella et al. 2010], this paper indeed describe the formal model and the PMS engine, whereas those papers mainly disseminate the project results and focus on the design and realization of the Task Handlers for PDAs according to a user-centered design approach.

The rest of the paper is organized as follows. Section 2 covers the state of the art in adaptability/flexibility in the leading PMSs. Section 3 describes the general framework by illustrating both the formalization of processes and the technique for automatically adapting such processes. Sections 4 and 5 describe how SmartPM has been realized in a proof-of-concept implementation. A specific example from a real-world process used in emergency management is developed in Section 6. Finally, Section 7 summarizes the contribution of the paper and outlines future work.

We have also introduced some appendixes. The first sketches some basic concepts on Situation Calculus and IndiGolog. The second gives an insight into a technique to translate process specifications given in WS-BPEL into the formal language used by SmartPM.

[2]Actually such a node could be a mobile robot, whose aim is to support the team of human operators.

Product	Manual	Pre-planned	Unplanned
YAWL [Adams et al. 2007]		✓	
COSA [Cosa GmbH 2008]	✓	✓	
Tibco [Tibco Software Inc. 2008]	✓	✓	
WebSphere [IBM Inc. 2008]	✓	✓	
SAP [Kinateder 2006]	✓	✓	
OPERA [Hagen and Alonso 2000]	✓	✓	
ADEPT2 [Göser et al. 2007]	✓		
ADOME [Chiu et al. 2000]	✓		
AgentWork [Müller et al. 2004]	✓		
ProCycle [Weber et al. 2009]	✓		
WASA [Weske 2001]	✓		
SmartPM			✓

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

2. RELATED WORK

Adaptation in PMSs can be classified in two main groups: evolutionary and exceptional changes. *Evolutionary changes* concern a planned migration of a process to an updated specification which, for instance, implements new legislations, policies or practices in business organizations, hospitals, emergency management, etc. Typically the inclusions of new evolutionary aspects are made manually by the process designer. When dealing with process specification changes, there is the issue of managing running instances, and, possibly, making migrate such instances to the updated specification [Sadiq et al. 2000; Casati and Shan 2001].

The survey [Rinderle et al. 2004] considers changes at the process schema and how adaptation of the process instances to the new schema is performed. It surveys many approaches wrt. completeness, correctness criteria and change realization (i.e., migration).

On the other side, there are the *exceptional changes* which are characterized by events, foreseeable or unforeseeable, during the process instance executions which may require instances to be adapted in order to be carried out. Since such events are exceptional, process specifications do not need any modifications. There are two ways to handling exceptional events. The adaptation can be *manual*: once events are detected, a responsible person, expert on the process domain, modifies manually the affected instance(s). The adaptation can be *automatic*: when exceptional events are sensed, PMS is able to change accordingly the schema of affected instance(s) in a way they can still be completed. Automatic adaptation techniques can be further broken down in two groups, pre-planned and unplanned, as previously introduced.

Table 2 compares the degree of adaptability to execeptional changes that is currently provided by the leading PMSs (either commercial or research proposals/prototypes). Among them, interesting approaches are ProCycle [Weber et al. 2009] and

ADEPT2 [Ly et al. 2008]. The first uses a case-based reasoning approach to support adaptation of workflow specifications to changing circumstances. Case-based reasoning (CBR) is the process of solving new problems based on the solutions of similar past problems: users are supported to adapt processes by taking into account how previously similar events have been managed. However, adaptation remains manual, since users need to decide how to manage the events though they are provided with suggestions. ADEPT2 features a check of semantic correctness to evaluate whether events can prevent processes from completing successfully. But the semantic correctness relies on some semantic constraints that are defined manually by designers at design-time and are not inferred, e.g., over pre- and post-conditions of tasks.

It is worthy to mention that during the 90s, [Heimann et al. 1996; Jaccheri and Conradi 1993] have addressed similar issues in the context of software development workflows, by adopting rewriting techniques in the first case and planning in the latter one.

Related to the issue of exceptional changes addressed with a pre-planned approach, there is the body of works that deal with compensations in PMSs. Indeed pre-planned approaches to exceptional changes (a.k.a. exceptions) are often based on the specification of exception handlers and compensation flows [Du et al. 1997; Casati et al. 1999; Hagen and Alonso 2000], with the challenge that in many cases the compensation cannot be performed by simply undoing actions and doing them again. In [Ellis and Keddara 2000], exception handlers are proposed as being workflow themselves. [Eder and Liebhart 1996] describes several types of compensation and provides a three-step mechanism to handle exceptions: *(i)* entails rollback based on the compensation type of activities in the workflow graph, then *(ii)* an agent (the one who initiated the rollback) determines whether to continue with the rollback, to take corrective measures, or to choose an alternative path, and finally forward execution is performed, which could lead to the same point of failure. [Golani and Gal 2005] describes how an optimal stop point can be detected (via formal analysis and interaction with the user), and how an alternative execution can be created automatically by bypassing the failed activity.

The SmartPM approach is complementary wrt. this literature, and leverages on it for dealing with exceptional changes that can be pre-planned. The novelty is that we propose, in addition to incorporating the previous techniques in a PMS, also to consider automatic adaptation to unplanned exceptions. In the following of the paper, we do not address the management of pre-planned exceptions, but it is intended to be incorporated in our proof-of-concept by adopting the previous techniques.

The scenario in which our approach has been conceived and validated is of pervasive nature. Not surprisingly adaptability is one of the very hot topics in the field of pervasive computing [Cetina et al. 2009; Garlan et al. 2002]; the novelty of our approach is that, unlike many other approaches in this field, it follows a process-oriented approach, which is particularly useful in the case of pervasive coordination of human actors. Finally we would like to mention that the usefulness of using a process-oriented approach for dealing with emergencies, as indirectly proposed by the WORKPAD project and this paper, is also confirmed by a number of research

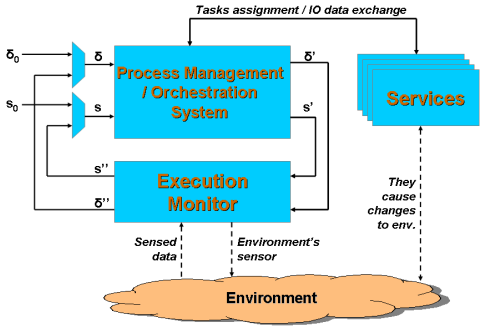

Fig. 1. Execution Monitoring in SmartPM

works, including [Franke et al. 2010; Wagenknecht and Rüppel 2009].

3. THE SMARTPM FRAMEWORK

The general framework proposed in this paper is based on the *execution monitoring* scheme described in [De Giacomo et al. 1998] for situation calculus agents. When using IndiGolog for process management, we take tasks to be predefined sequences of actions and processes to be IndiGolog programs. IndiGolog [De Giacomo et al. 2009] is a programming language that relies on a set of actions defined in a certain Domain Theory. Domain Theories define the properties of interest of the world (in our case process domain), and one can define the pre- and post-conditions of actions on the basis of these properties[3]. Before a process starts, SmartPM takes the initial context from the real environment and builds the corresponding knowledge base (i.e., set of logic formulas) corresponding to the initial situation S_0. SmartPM also builds an IndiGolog program δ_0 corresponding to the process to be carried on. Then, at each execution step, SmartPM, 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 whose preconditions are fulfilled. A service can collect data needed to execute the task assigned by SmartPM, and when a service finishes executing a task, it alerts SmartPM. The execution of a process can be interrupted by the *monitor module* when a misalignment between the virtual and the physical realities is discovered. In that case, the monitor *adapts* the (current) program to deal with such discrepancy. In Figure 1, the overall framework is depicted. More specifically, at each step, SmartPM advances the current process δ in the current situation s by executing an action, resulting then in a new situation s' with the process δ' remaining to be executed. Both δ' and s' are passed to the monitor, which is also meant to collect data from the environment through *sensors*. If a discrepancy between the virtual reality (as represented by what holds true in situation s') and the physical reality is sensed, then the monitor first changes/extends situation s' to a new situation s'', by synthesizing a sequence of actions that explains the changes

[3]The reader interested in further details can refer to Appendix A. The appendix is here included for reviewers' convenience. If hopefully accepted, and depending on the final page limit, it will be included or not in the paper.

perceived in the environment, thus re-aligning the virtual and physical realities. Unfortunately, though, it could be the case that the current remaining process δ' may not be able to execute successfully anymore (i.e., assign all tasks as required) in the new (unexpected) situation s''. If so, the monitor then also adapts the (current) process by performing suitable recovery changes and generating a new remaining process δ''. At this point, the main process is resumed and the execution continues with program-process δ'' in situation s''.

Let us now see in detail how all this is technically achieved.

3.1 Process Formalization

Here, we explain how processes are realized in the Situation Calculus. First of all, our formalization makes use of the following domain-independent predicates (that is, non-fluent rigid predicates) to denote the various objects of interest:

—$Service(srvc)$: $srvc$ is a service;

—$Task(task)$: $task$ is a task;

—$ListElem(workitem(task, input))$: it denotes all admissible objects $workitem(task, input)$ which are specifically pairs composed of a task $task$ and $input$;

—$Capability(b)$: b is a capability;

—$Provides(srvc, b)$: service $srvc$ provides capability b;

—$Requires(task, b)$: task $task$ requires the capability b;

Also, to refer to the ability of a certain service $srvc$ to perform a certain list of work items $\vec{l}$, we introduce term "work-list" and define the following abbreviation:

$$Capable(srvc, \vec{l}) \equiv \\ [\forall t, b.\ workitem(t, input) \in \vec{l} \wedge (Requires(b, t) \Rightarrow Provides(srvc, b))].$$

That is, service $srvc$ can carry out a certain work-list $\vec{l}$ iff $srvc$ provides all capabilities required by every work-item's task t in $\vec{l}$. The concept of work-list has been introduced with the purpose of constraining a list of (sequential) work items to be executed by a single service.

The execution of a work-list involves the execution of four basic actions:

—$assign(srvc, \vec{l})$: a work-list $\vec{l}$ is assigned to a service $srvc$; this means $srvc$ is assigned to execute every task of work-list.

—$start(srvc, t, p)$: service $srvc$ is notified to be allowed to start task t with input values p;

—$ackCompl(srvc, t)$: service $srvc$ acknowledges of the successful completion of task t;

—$release(srvc, \vec{l})$: the service $srvc$ is released with respect to work-list $\vec{l}$.

When SmartPM executes an action, all services are notified, but services that are not interested in that action just ignore the notification. The actions performed by SmartPM are meant to be "complemented" by actions executed by the services themselves. Such actions are meant to inform the SmartPM engine on how task executions are progressing and are of two types:

—*readyToStart*$(srvc, t)$: service $srvc$ declares to be ready to start performing task t.

—*finishedTask*$(srvc, t, q)$: service $srvc$ declares to have completed the execution of task t with output q.

After a service $srvc$ is assigned to a work-list, tasks must be started and completed in the sequential order denoted by the work-list itself.

Informally, for every task t, $srvc$ needs to report to be ready to start it through executing *readyToStart*$(srvc, t)$. For this to happen, the service needs to have sufficient resources for carrying out the task in question. Only after a service reports to be ready, SmartPM can eventually perform action *start*$(srvc, t, p)$, which instruct SmartPM to begin the execution. Terms p and q denote arbitrary sets of inputs/outputs, which depend on the specific task; special constant $\emptyset$ denotes empty input/output. After a service reports that the task has been completed by the execution of action *finishedTask*$(srvc, t, q)$, SmartPM can update the value of fluents according to the successful execution of task t with output q. In addition, SmartPM acknowledges the completion, via action *ackCompl*$(srvc, t)$.

The actions from *readyToStart*$(srvc, t)$ to *ackCompl*$(srvc, t)$ are repeated for all tasks in the list. Once they all are completed, SmartPM release the service from the work-list, via action *release*$(srvc, \vec{l})$.

When it comes to specify processes in IndiGolog, two classes of fluents are used. The first class includes those fluents which are used to manage the task life-cycle and the resource perspective of processes. Apart from one fluent, which needs to be customized for every domain, the definition of the others are domain-independent and so do not change across different domains. The second class concerns such fluents used to denote the data structure of process instances; their definitions do depend on the specific process domain of interest.

Fluents for task life-cycle management. Task assignment is driven by fluent *Free*$(srvc, s)$, which is independent of the process domain and whose intended meaning is that service $srvc$ is free in situation s. Its successor state axiom is as follows:

$$\begin{aligned} &\mathit{Free}(srvc, do(a, s)) \equiv \\ &\quad \big(\exists q.a = \mathit{release}(srvc, q)\big) \vee \big(\mathit{Free}(srvc, s) \wedge \neg\exists p.a = \mathit{assign}(srvc, p)\big). \end{aligned} \tag{1}$$

Service $srvc$ is considered free in the current situation iff it has just been released or it was free in the previous situation and no task has been assigned to it.

The fact that a certain service $srvc$ is free does not directly imply that it can be assigned to a task. In the scenario sketched in the introduction, for example, a service, in order to be assigned a task, should be free as well as connected to the leader. So, fluent *Free*$(srvc, s)$ is not directly used to determine whether a certain service $srvc$ may be assigned to a task. To that end, the framework relies on predicate *Available*$(srvc, s)$, which states whether a service $srvc$ is available in situation s for task assignment:

$$\mathit{Poss}(\mathit{assign}(srvc, \vec{l}), s) \Rightarrow \mathit{Available}(srvc, s).$$

Unlike *Free*$(srvc, s)$, this predicate is meant to be specifically defined for each domain; all that is required is that its definition (that is, successor state axiom) must

enforce the following constraint:

$$\forall srvc.Available(srvc, s) \Rightarrow Free(srvc, s). \tag{2}$$

Availability of a service is insufficient from being assigned to a task: a service is required to provide every required capability . Please note that *Available* is not a fluent, since its value is not modified as direct consequence of action performances. It is simply a shortcut that is, in general, a disjunction of fluents.

Data Fluents. A family of data fluents is used to store the outcome of tasks. The values of these fluents can be passed as inputs to services for task performances as well as used in the expressions at decision points (e.g., for cycles, conditional statements) in processes. In addition, the values of a data fluent can be passed to another data fluent, i.e., "copied." So, if X is a process data variable that is meant to capture the outcomes of (specific) task t $\in T_1, \ldots, T_n$, then the situation calculus domain theory shall include a functional fluent $X(s)$ with the following successor state axiom:

$$\begin{array}{l} X(do(a,s)) = y \equiv \\ \quad [\exists srvc, t.a = \mathit{finishedTask}(srvc, t, y) \wedge t \in \{T_1, \ldots, T_n\}] \vee \\ \quad [a = Copy^{X}_{Y_1} \wedge Y_1(s) = y] \vee \cdots \vee [a = Copy^{X}_{Y_n} \wedge Y_n(s) = y] \vee \\ \quad [X(s) = y \wedge \\ \quad (\neg\exists srvc2, t', y'.a = \mathit{finishedTask}(srvc2, t', y') \wedge t' \in \{T_1, \ldots, T_n\}) \wedge \\ \quad [a \neq Copy^{X}_{Y_1} \wedge \cdots \wedge a \neq Copy^{X}_{Y_n}]] \end{array} \tag{3}$$

where Y_i are all the data fluents different from X, and each action $Copy^{X}_{Y}$, always possible, is meant to "copy" the value of data fluent $Y(s)$ into fluent $X(s)$. The value of data fluent $X(s)$ is changed to value y when one of the corresponding tasks finishes with output y or $X(s)$ is assigned the value of another data fluent whose current value is y.

Finally, the theory contains axioms describing the initial situation S_0: all services are assumed free and not reserved for any task. As far as Data Fluents, their initial value should be defined manually at design-time, as these fluents depend on the specific domain.

3.2 The Adaptability Technique

We now turn our attention to how monitoring and adaptation are meant to work within the process formalization given in the previous section. Intuitively, from the current program δ and the current situation s from the PMS's virtual reality, the monitor may first build a new situation s' from s by introducing new "explanatory" actions that align the virtual reality of the PMS with sensed information. The monitor then analyzes whether δ can still be executed in s', and if not, it adapts δ into an an alternative executable program δ'. Technically, the monitor can be formally defined as follows (we do not model how the situation s' is generated from the sensed information):

$$\begin{array}{l} \mathit{Monitor}(\delta, s, s', \delta') \equiv \\ \quad [\mathit{Relevant}(\delta, s, s') \wedge \mathit{Recovery}(\delta, s, s', \delta')] \vee [\neg \mathit{Relevant}(\delta, s, s') \wedge \delta' = \delta], \end{array} \tag{4}$$

where *(i)* $Relevant(\delta, s, s')$ states whether the changes from situation s to s' are such that δ may not correctly execute anymore; and *(ii)* $Recovery(\delta, s, s', \delta')$ is intended to hold whenever program δ' is an adaptation of program δ, to be now executed in situation s' instead of s.

Relevant can be given in different forms but in a way that complies with the following

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

where $SameConfig(\delta, s, \delta', s')$ holds if executing δ from situation s is "equivalent" to executing δ' from situation s'. In other terms, it holds if it denotes any type of bisimulation [Milner 1980].

Currently, SmartPM adopts $SameConfig(\delta, s, \delta', s') \stackrel{\text{def}}{=} SameState(s, s')$; therefore:

$$Relevant(\delta, s, s') \stackrel{\text{def}}{=} \neg SameState(s, s'). \tag{5}$$

Here, predicate $SameState(s's')$ holds iff the states in situations s' and s are the same. Observe such predicate can actually be defined as a first-order formula as the conjunction of formulas $F(s) \equiv F(s')$ for each fluent F defined in the action theory.

When it comes to the actual recovery, we formalize predicate $Recovery(\delta, s, s', \delta')$ for adapting a given process δ as follows:

$$\begin{aligned} &Recovery(\delta, s, s', \delta') \equiv \\ &\quad \exists \delta_a . \delta' = \delta_a; \delta \wedge Linear(\delta_a) \wedge Do(\delta_a, s', s'') \wedge \neg SameState(s', s''). \end{aligned} \tag{6}$$

Given a process δ and situations s and s', the following results state the soundness and completeness of the approach.

THEOREM 3.1.

$$\exists \delta_b . Do(\delta_b, s', s'') \wedge SameState(s', s'') \equiv \exists \delta' . Recovery(\delta, s, s', \delta')$$

PROOF. The only difference between the two definitions is that in the second case (the right part) we allow only for linear programs (i.e., sequences of actions) as δ_a, while in the first case (the left part) any deterministic program is allowed, which may include also cycles, `if-then-else`, etc.
($\Leftarrow$) Trivial, as linear programs are also deterministic ones.
($\Rightarrow$) Program δ_b is not a linear program, but actions are atomic. As such, the actual execution is an interleaving of the different branches that δ_b is composed by. Let us assume that the interleaving produces the sequential execution of n actions $a_1, a_2, \ldots, a_n$. Therefore, it follows that:

$$s'' = (a_n, do(a_{n-1}, \ldots, do(a_2, do(a_1, s'))$$

Let us consider the linear program $\vec{p} = (a_1, a_2, \ldots, a_n)$. Since $Do(p, s', s'')$ evaluates obviously true, $Recovery(\delta, s, s', p)$ holds. □

THEOREM 3.2. *Assume a domain in which services as well as input and output parameters are finite. Computing a process δ' such that $Recovery(\delta, s, s', \delta')$ holds is decidable.*

PROOF. In domains where services and input and output parameters are finite, ground concrete actions and fluents are also finite. Hence we can phrase the domain as a propositional planning problem which is known to be decidable [Ghallab et al. 2004]. □

Thus, to adapt process δ, one needs to determine a *linear* program δ_a (i.e., a sequence of actions) that would bring the virtual reality to a situation s'' whose corresponding state is the same one as that of s'. Observe that this specification asks to search for a linear program that achieves a certain formula, namely, $SameState(s', s'')$. As a result, we have reduced the synthesis of a recovery program to a planning problem in AI [Ghallab et al. 2004], for which efficient techniques exist in the literature. In Section 4, we shall see how the current proof-of-concept implementation of SmartPM uses the lookahead search construct Σ provided by IndiGolog to solve such planning problems. Note that finite domains is not a strong limitation in our setting, since it is possible to discretize the admissible values without losing much detail.

4. REALIZING THE FRAMEWORK

In Figure 2, we show how SmartPM has been concretely coded by the interpreter of IndiGolog[4]. The main procedure involves two concurrent programs in priority. The monitor, which runs at higher priority, is in charge of monitoring changes in the environment and adapting accordingly. At a lower priority, the system runs the actual IndiGolog program representing the process to be executed, namely procedure **Process**().

This procedure relies, in turn, on procedure **ManageExecution**, which includes task assignment, start signaling, acknowledgment of completion, and final release. In Figure 2 we make use of a Prolog-like notation: lists are enclosed between squared brackets and notations like [*elem* | *tail*] denote lists composed by concatenating a first element *elem* with a tail list *tail*.

The first step in procedure **Monitor** checks whether fluent *Exogenous* holds true, meaning that an exogenous (unexpected) action has occurred in the system. As a result, when no exogenous events has yet occurred, the procedure is stuck on that test, i.e., it cannot perform a step. It is only then when the business process of interest (i.e., program **Process**) may execute/advance.

Now, consider the situation in which an exogenous action has indeed occurred and so fluent *Exogenous* holds true. Then, procedure **Monitor** is enabled and it performs a step on line 1. After that the monitor checks whether the exogenous event is relevant, in that they require some kind of adaptation. If so, then the monitor triggers the actual adaptation module. Specifically, the monitor searches for a full execution of program **Adapt**, which will build the recovery program/process. Finally, the last step of the monitor involves resetting fluent *Exogenous* to false, by executing action *resetExog*.

Recall that, from Equation (4), an exogenous event is deemed relevant for adaptation if it yields a different state than the one expected. To do so, the definition compares the state in the current situation s with the new (unexpected) situation

[4] `http://sourceforge.net/projects/indigolog/`

```
Proc Main()
1  ⟨Exogenous ∧ ¬Finished → [Monitor()]⟩;
2  ⟨true → [Process(); finish]⟩;
3  ⟨¬Finished → [wait ]⟩;

Proc Monitor()
1  if Relevant
2     then Σ([Adapt()], assumptions[{assign(srvc, [task]), readyToStart(srvc, task)},
3               {start(srvc, task, p), finishedTask(srvc, task, p)}]);
4  resetExog;

Proc Adapt()
1  Plans(0, 10);

Proc Plans(m, n)
1  (m ≤ n)?;
2  [ActionSequence(m); (¬Relevant)?] | Plans(m + 1, n);

Proc ActionSequence(n)
1  if n > 0
2     then π(task, x); ListElem(workitem(task, x))?;
3          ManageExecution([workitem(task, x)]);
4          ActionSequence(n − 1);

Proc ManageExecution(workList)
1  π(srvc); (Capable(srvc, workList) ∧ Available(srvc))?;
2  assign(Srvc, workList);
3  if (workList ≠ [ ])
4     then ExecutionHelp(Srvc, workList);
5  release(Srvc, workList);

Proc ExecutionHelp(Srvc, [workitem(Task, Input) | TailWorkList])
1  start(Srvc, Task, Input);
2  ackCompl(Srvc, Task);
3  if (TailWorkList ≠ [ ])
4     then ExecutionHelp(Srvc, TailWorkList);
```

Fig. 2. The core procedures of SmartPM.

s'. However, the IndiGolog interpreter always evaluates formulas on the *current* situation only; there is no way to get access to past situations. Fortunately, however, we can get around this limitation for our specific needs as follows. For each fluent $F(\vec{x}, s)$, we introduce a new fluent $F_{prev}(\vec{x}, s)$ that is meant to record the "old" value of F. The successor state axiom for the new fluent is straightforward:

$$F_{prev}(\vec{x}, do(a, s)) = p \equiv$$
$$(\neg ExogAction(a) \wedge F(\vec{x}, s) = p) \vee (F_{prev}(\vec{x}, s) = p \wedge ExogAction(a)).$$

That is, F_{prev} records the (current) value of fluent F just after action a provided a is not an exogenous event itself. With these fluents defined, one can then define fluent *Relevant* as the following abbreviation:

$$Relevant(s) \stackrel{\text{def}}{=} \bigvee_{F \in \Delta} \neg\big(F(\vec{x}, s) \equiv F_{prev}(\vec{x}, s)\big).$$

where Δ is the (finite) set of fluents defined in the action theory.

Finally, let us focus on the actual program in charge of recovery, namely, procedure **Adapt**. Generally speaking, such procedure will try to reach a situation in which *Relevant* is not true anymore—see test in line 2. To do so, the recovery procedure "simulates" the performance of zero, one, or more tasks in compatible

services (line 1). Every single task is included in a separate work list composed of one work item; hence, every task list created by the recovery contains exactly one task.

Observe that **Adapt** uses a simple breadth-first planning mechanism, which specialized what proposed in [Reiter 2001]. It returns shortest plans and during the testing phase has been proven to be faster, on average. The planner first generates the actions (i.e., 4 actions: *assign*, *start*, etc.) to execute one of the available tasks and, then, checks if any of such sequences makes recover from the deviation. Otherwise, it generates the actions to execute a length-two sequence of tasks. The planning technique is iterative deepening: if there exists no sequence whose length is less or equal to n tasks, it tries with length-$(n+1)$ task sequences[5].

Observe that, the non-deterministic choice (i.e., in the form of $(\delta_1|\delta_2)$) does not specify how many tasks are to be performed or even which ones on which services. Such details are left to the *automated planner*, that is, search operator Σ, to resolve at execution time. A successful execution would need to resolve all the non-deterministic choices (i.e. the sequence length and the services and tasks picked) in a way that would guarantee that condition *Relevant* would not hold true anymore.

In fact, we use a specialized version of search operator Σ that relies on some assumptions on the performable actions. Every assumption is of form $\{actionPMS(\vec{x}), actionService(\vec{y})\}$, meaning that action *actionPMS* executed by the SmartPM engine with input $\vec{x}$ will be eventually "complemented" by action *actionService* executed by a service with input $\vec{y}$. Vector of parameters $\vec{y}$ is a fully-deterministic transformation of $\vec{x}$, which is accordant to a certain expression. Here we are using the simple case where $\vec{y}$ is identically equal to $\vec{x}$, but one can customize for specific tasks/actions. The obvious question that arises is what happens if assumptions are not held (e.g., $assign(srvc, [task])$ is followed by a certain action $finishedTask(srvc, task, p)$ or a new exogenous event). In that case, the built recovery plan is considered as failed and a new one is built again starting from the new situation.

It is worth highlighting that, because the monitor runs at a higher-priority level than the actual process, the solution plan found for the recovery program $\Sigma[\mathbf{Adapt}]$ would run at higher-priority than program **Process**. So, the process program **Process** cannot progress until the recovery is finished. Consequently, after a sensed deviation, the program executed would be equivalent to $(\Sigma[\mathbf{Adapt}()]; \delta')$, where δ' is the program remaining from procedure **Process**, thus matching exactly the definition of Equation (6).

5. THE SMARTPM PROTOTYPE

This section aims at describing the internal structure of SmartPM. Figure 3 shows its conceptual architecture. At the beginning, a responsible person designs the process in the form of a WS-BPEL file plus some XML notations. Indeed process

[5]This keeps going deeper and deeper till reaching a sequences of 10 tasks or any task sequence that recovers. If no task sequence of at most 10 tasks exists, it is assumed that no recovery is possible. We have adopted 10 as bound to the length of the sequence as it is a reasonable assumption in our scenario.

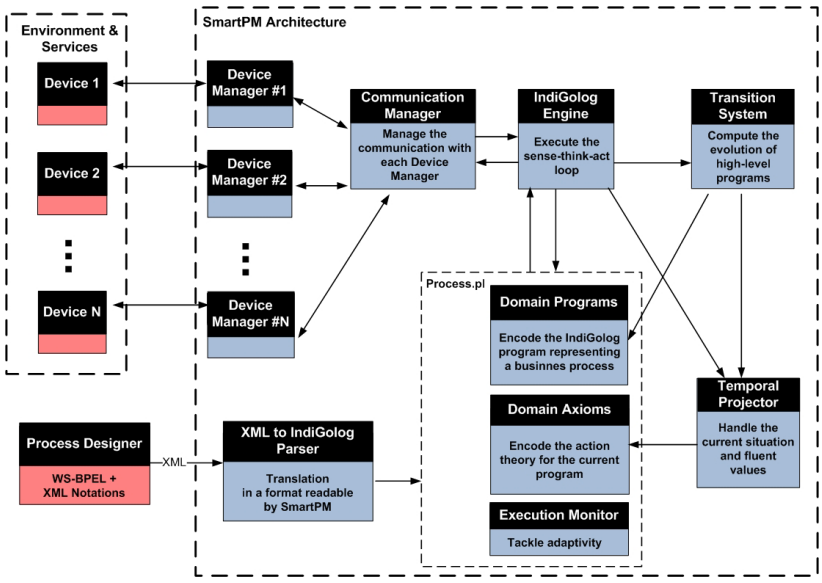

Fig. 3. Architecture of SmartPM.

designers are not intended, in our approach, to define processes using directly the SmartPM framework. Conversely, on the basis of the requirements stemming from the WORKPAD project, they specify the processes in WS-BPEL, and have available a kind of templates (specific for different scenarios) for specifying "semantic aspects" needed in our approach (e.g., pre- e post-conditions, service capabilities and required skills for a task, etc.). In the future, we aim to consider other (conceptual) languages for process specifications, in particular BPMN – Business Process Modeling Notation – and YAWL.

Then, such files are loaded into SmartPM. The *XML-to-IndiGolog Parser* component translates this specification in a *Domain Program*, the IndiGolog program corresponding to the designed process, and a set of *Domain Axioms*, which define the initial situation and the set of available actions with their pre- and post-conditions. The parser generates also a set of static routines, collectively named *Execution Monitor*, that code the adaptation features (see Section 4)

When the program is translated in the Domain Program and Axioms, a component named *Communication Manager* (CM) starts up all of *device managers*, used by SmartPM to communicate with the services and sensors installed on devices. For each real world device SmartPM holds a device manager. Each device manager is also intended for notifying the corresponding device of every action performed by the SmartPM engine as well as for notifying the SmartPM engine of actions executed by the services of the corresponding device.

After this initialization process, CM activates the *IndiGolog Engine*, which is in charge of executing IndiGolog programs. The IndiGolog Engine executes a *sense-think-act* interleaved loop [Kowalski 1995], cycling continuously the following three steps:

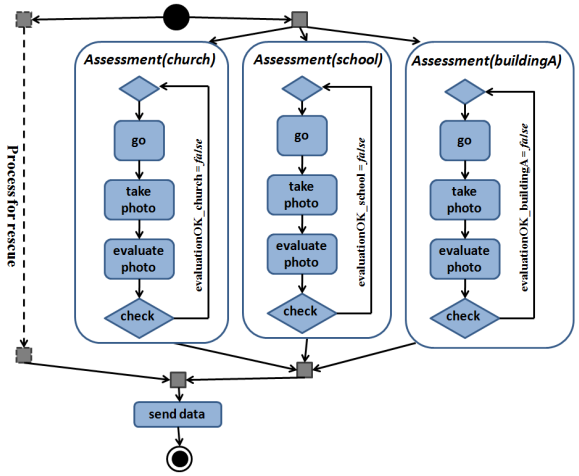

Fig. 4. An example from emergency management

(1) check for exogenous events that have occurred;
(2) calculate the next program step; and
(3) if the step involves an action, *execute* the action, instructing the Communication Manager.

The IndiGolog Engine relies on two further modules named *Transition System* and *Temporal Projector*. The former is used to compute the evolution of IndiGolog programs according to the statements' semantic, whereas the latter is in charge of holding the current situation throughout the execution as well as letting evaluate the fluent values for taking the right decision of the actions to perform.

6. A CONCRETE EXAMPLE FROM EMERGENCY MANAGEMENT

This section is aimed to show the application of SmartPM in a concrete case stemming from WORKPAD. The example in Figure 4 depicts a (part of a) process concerning the management of the aftermath of an earthquake. It is composed by two main branches: the left, abstracted out for space reasons, concerns saving people trapped under collapsed buildings, whereas the right branch is intended to assess the zones that require a more thorough analysis. Let us focus on the right branch.

For each critical point of interest (POI) of the affected area, an entire sub process is enacted. For the sake of simplicity, we abstract it out as a sequence of three tasks. The first two tasks are constrained to be executed by the same person and concern moving to the POI and, then, taking some photos of such a POI. Finally, the next task is to evaluate the photos taken and is in general executed by a different person. If the evaluation is unsuccessful (i.e., the photos' quality is not enough), new photos need to be taken. Once all sub processes for every POI are carried on, there is a synchronization which is followed by task "send data" that is intended to send all the collected information to some headquarter. Table II shows the core procedures, predicates and fluents that map the activity diagram above.

SITUATION CALCULUS PREDICATES, SHORTCUTS AND PRE-CONDITIONS

$Service(Srv1)$.
$Service(Srv2)$.
$Service(Srv3)$.
$Service(Srv4)$.
$Service(Srv5)$.

$Task(Go)$.
$Task(TakePhoto)$.
$Task(EvaluatePhoto)$.
$Task(SendData)$.

$Capability(Camera)$.
$Capability(Evaluation)$.
$Capability(GPRS)$.

$Provides(Srv1, Evaluation)$.
$Provides(Srv1, GPRS)$.
$Provides(Srv2, Camera)$.
$Provides(Srv3, Evaluation)$.
$Provides(Srv4, Camera)$.
$Provides(Srv4, Evaluation)$.
$Provides(Srv5, Camera)$.

$Requires(TakePhoto, Camera)$.
$Requires(EvaluatePhoto, Evaluation)$.
$Requires(SendData, GPRS)$.

$PoiType(StartingPoint)$.
. . . . . .
$PoiType(Church)$.

$Location(o) = l \equiv$
$o = Loc(x, y) \wedge$
$integer(x) \wedge (x >= 0) \wedge (x <= 10) \wedge$
$integer(y) \wedge (y >= 0) \wedge (y <= 10)$.

$PoiLocation(StartingPoint, Loc(0, 0))$.
. . . . . .
$PoiLocation(Church, Loc(5, 6))$.

$Poss(start(srvc, TakePhoto, p), s) \equiv$
$At(srvc, s) = p$.

$Available(srvc, s) \equiv$
$Free(srvc, s) \wedge Connected(srvc, s)$.

$Connected(srvc, s) \equiv$
$Neigh(srvc, Srv1, s) \vee$
$\exists srvc2.(Neigh(srvc, srvc2, s) \wedge Connected(srvc2, s))$.

$Neigh(srvc1, srvc2, s) \equiv$
$At(srvc1, s) = p \ \wedge At(srvc2, s) = q$
$\wedge ||q - p|| < range$.

(a)

SITUATION CALCULUS FLUENTS

$At(srvc, do(a, s)) = l \equiv$
$[a = finishedTask(srvc, Go, p) \wedge PoiType(p) \wedge$
$PoiLocation(p, l)] \bigvee$
$[At(srvc, s) = l \ \wedge \neg\exists srvc, l', p'.Location(l') \wedge PoiType(p') \wedge$
$PoiLocation(p', l') \wedge a = finishedTask(srvc, Go, p')]$.

$PhotoBuild\_Church(do(a, s)) = true \equiv$
$[\exists srvc.a = finishedTask(srvc, TakePhoto, V) \wedge V = Church] \bigvee$
$[PhotoBuild\_Church(s) = true \wedge$
$\neg\exists srvc.a = finishedTask(srvc, TakePhoto, V') \wedge V' = Church]$.

$EvaluationOK\_Church(do(a, s)) = true \equiv$
$[\exists srvc.a = finishedTask(srvc, EvaluatePhoto, V) \wedge V = Church] \bigvee$
$[EvaluationOK\_Church(s) = true \wedge$
$\neg\exists srvc.a = finishedTask(srvc, EvaluatePhoto, V') \wedge V' = Church]$.

. . . . . .

INDIGOLOG PROCESS

```
PROC Process()
 1  (TaskFlow(); InvokeSendData())

PROC TaskFlow()
 1  (BranchA() ‖ BranchB())

PROC BranchA()
 1  ...processes for rescue...

PROC BranchB()
 1  (BranchB1() ‖ BranchB2() ‖ BranchB3())

PROC BranchB1()
 1  while ¬(EvaluationOK_Church)
 2  do SequenceB1()

PROC SequenceB1()
 1  (InvokeGoChurch();
 2  InvokeTakePhotoChurch();
 3  InvokeEvaluatePhotoChurch())

PROC InvokeGoChurch()
 1  [ManageExecution(Go, church)]

PROC InvokeTakePhotoChurch()
 1  [ManageExecution(TakePhoto, church)]

PROC InvokeEvaluatePhotoChurch()
 1  [ManageExecution(EvaluatePhoto, church)]

. . . . . .

PROC InvokeSendData()
 1  [ManageExecution([SendData, input])]
```

(b)

Table II. The code of the example (parts). The complete version is available at `http://www.dis.uniroma1.it/~marrella/public/TAAS/appendixes.zip`

On the left-hand side, the table shows the predicates to define 5 services, 4 tasks, 3 capabilities as well as to specify the capabilities that services provide and tasks require. In addition, there exists predicate *PoiType* which defines the POIs that are significant for the process domain. Predicate *Location* gives the definition of the locations where services can be located and tasks can be executed. Once predicate *PoiLocation* has been introduced, it is very important to associate POIs to locations and predicate *PoiLocation* is exactly meant for this. Furthermore, the left-hand side of the table shows an example of attaching pre-conditions to tasks. Specifically, the following definition is intended to constrain service *srvc* that starts task $TakePhoto$ to be located in the location p where task needs to be executed:

$$Poss(start(srvc, TakePhoto, p), s) \equiv At(srvc, s) = p$$

The lower part of Table II(a) includes some additional definitions. The first concerns $Available(srvc, s)$; in this example, a service *srvc* is available to be assigned to a task, if it is currently assigned to no task and, moreover, the device that hosts service *srvc* is connected to the team network. To handle this last requirement, a procedure $Connected(srvc, s)$ is defined. It specifies that a service *srvc* is connected to the team network if either it is in the radio range of device *Srv*1 (the service elected as Coordinator), that deploys the SmartPM engine, or it is in proximity of a further service *srvc*2 connected to the network (cf. the procedure *Neigh*).

Table II(b) starts showing the domain-dependent fluents used to capture the state of the process instance. In particular, since the process depicted in Figure 4 requires to take some photos and evaluate them in three different POIs - respectively called *Church*, *School* and *BuildingA* - the definition of three couple of fluents to record these aspects are needed. For space reasons, here we refer only to those fluents provided for POI identified as *Church*. Therefore, concerning the domain-dependent fluents shown in the Table II(b), one can easily identify their successor state axiom, following:

—$At(Srvc, s) \equiv q$ evaluate to a value q in the domain of predicate *Location*. It is aimed to represent the position of *srvc* in situation s.

—$PhotoBuild\_Church(s) \equiv true$ if some pictures have been taken in that POI identified as *Church*.

—$EvaluationOK\_Church(s) \equiv true$ if some pictures have been evaluated successfully in that POI identified as *Church*.

Finally, the lower part of Table II(b) shows the SmartPM program of the process, which describes the control flow of tasks to be executed.

The section concludes with delineating an example of recovering from a deviation applied to the process define above. Please note that in the following we will not describe the policies to recover from deviations, as we would fall into the case of pre-planned adaptation. SmartPM will reason over the current situation to devise the most opportune recovery plan without any a-priori policy. For the sake of explanation, let now assume to have defined an exogenous event $photoLost(p)$ where p is a specific POI. This exogenous event models the event when some photos, previously taken in a specific POI p, get lost (e.g., due to the unwilling deletion of some files). Consequently, we need to change the successor-state axiom for fluent

PhotoBuild to model the effect of *photoLost* onto this fluent (now we are supposing that p=*Church*) :

$$\begin{array}{l} \mathit{PhotoBuild\_Church}(do(a,s)) = \mathit{true} \equiv \\ \quad [\exists \mathit{srvc}.a = \mathit{finishedTask}(\mathit{srvc}, \mathit{TakePhoto}, V) \wedge V = \mathit{Church}] \bigvee \\ \quad [\mathit{PhotoBuild\_Church}(s) = \mathit{true} \ \wedge a \neq \mathit{photoLost}(\mathit{Church}) \wedge \\ \quad (\neg \exists \mathit{srvc}.a = \mathit{finishedTask}(\mathit{srvc}, \mathit{TakePhoto}, V') \wedge V' = \mathit{Church})]. \end{array}$$

Suppose now that in current situation $\overline{s}$, $\mathit{PhotoBuild\_Church}(\overline{s}) = \mathit{true}$, i.e. some photos have been previously taken in that *Location* associated to the *Church*. In this situation, exogenous event *photoLost*(*Church*) occurs (i.e., in the POI identified as *Church* some photos get lost). Now, SmartPM should find a recovery program which restores the previous value for the fluent. After performing some testing, we experienced SmartPM finds the following recovery program, where two tasks need to be executed by a certain service *srvc* (assuming $At(\mathit{srvc}, \overline{s}) \neq l$, where l is the *Location* in which the *Church* is located):

ManageExecution(*Go*, *srvc*, `Church`)
ManageExecution(*TakePhoto*, *srvc*, `Church`)

The second task is what actually restores the value expected. But, in order for *srvc* to take photo in the POI selected, it needs to be already close to that location. This is the reason why task *Go* precedes *TakePhoto* in the program to recovery the photos lost. Furthermore, before to assign the above two tasks, SmartPM engine is also able to estimate if *srvc* will disconnect itself from the network during the path covered to reach the *Church* (it happens, in particular, if the *Church* is far away from the current position of the Coordinator). In such a case, to prevent any disconnection, it is required the assignment of a further task *Go* to another service *srvc*2 that is both available and in the radio range with the Coordinator :

ManageExecution(*Go*, *srvc*2, `p'`)

Since the main purpose is to guarantee the maintenance of the network, *srvc*2 should act as a "bridge"; it must reach a specific *POI* p' chosen on the fly by the engine and well-located (according to the strength and to the range of the network) in a position between the leader and service *srvc*. Once the photos have been restored, the process can progress again.

In order to prove our approach, we performed some testing to learn the time amount required to build the recovery plan over the process shown in the example. The x-axis of the chart in Figure 5 shows the seconds needed to find a recovery plan of a specific length in a area in which, respectively, 20, 30 and 40 POIs have been fixed (cf. y-axis). The length of a plan is directly linked to the number of tasks required to handle the adaptation. As an instance, in the example seen above, the SmartPM engine found a 3-length recovery plan. For each of the three sets of POIs specified, we ran 10 different tasks causing a relevant deviation from the original process schema. We performed these tests in order to obtain, respectively, 1- , 2- and 3-length plans. The chart in Figure 5 captures the mean-time obtained by these 10 executions on a netbook. By analyzing collected time values, it is clear that increasing the number of POIs in the map - in other words, increasing the size of the input - the time needed to find a plan of a specific length grew

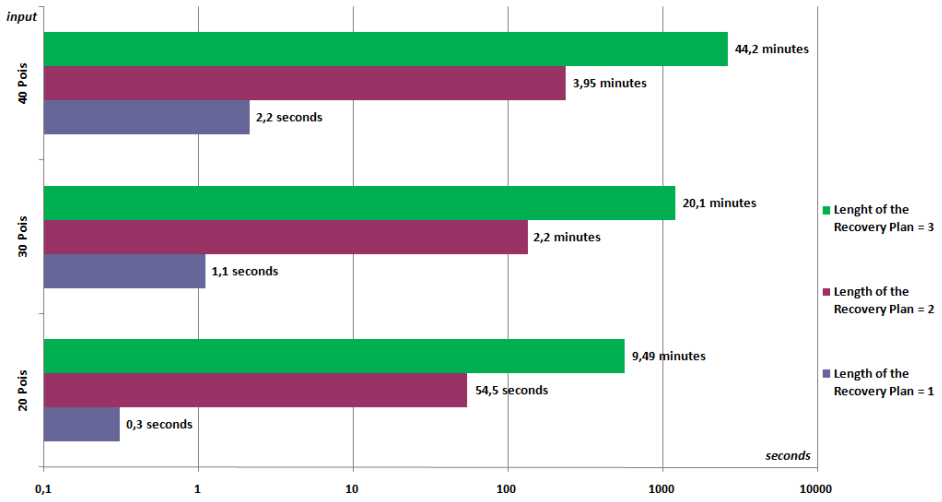

Fig. 5. Time needed to build a recovery plan in the example

exponentially. This kind of results could be expected, being the adaptation based on the solution to a planning problem, which has EXPTIME complexity. However, results obtained for a map composed by 20 POIs (that fit well a real case), are quite acceptable. Moreover, we report such tests to show the practicability of the approach. An engineered solution would adopt state-of-art planners, more efficient in most causes, and not the naive `IndiGolog` implementation.

7. CONCLUSIONS

Most of existing PMSs are not completely appropriate for highly dynamic and pervasive scenarios. Such scenarios are turbulent and subject to a higher frequency of unexpected contingencies with respect to usual business settings, which show a static and foreseeable behavior. This paper describes SmartPM, a (model and a prototype of) PMS that is able to automatically adapt processes by recovering them from exceptions, without relying either on the intervention of domain experts or on the existence of pre-specified handlers to cope with specific exceptions.

Future work aims mostly at integrating SmartPM with state-of-art planners. Indeed, current implementation relies on the IndiGolog built-in planner, which performs a blind search. Current planners make use of advanced techniques for reducing the search space. A challenging issue then is how to convert action theories and IndiGolog programs into the inputs of an automated planner (e.g., by translating to PDDL).

Moreover, we would like to investigate how to remove the limitation of linear recovery plans, on the basis of the results we gained in [de Leoni et al. 2009]. Finally we intend to show how to map all workflow patterns to the SmartPM model, in order to create a library of predefined process schemas and related annotations to be applied in concrete scenarios.

Acknowledgement.. Special thanks to Giuseppe De Giacomo for many useful technical discussions and the continuous support to this research. Thanks also to Arthur H.M. ter Hofstede for his suggestions improving the paper presentation.

REFERENCES

Adams, M., ter Hofstede, A. H. M., van der Aalst, W. M. P., and Edmond, D. 2007. Dynamic, extensible and context-aware exception handling for workflows. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS Proceedings, Part I*. Lecture Notes in Computer Science, vol. 4803. Springer, 95–112.

Battista, D., de Leoni, M., Gaetanis, A., Mecella, M., Pezzullo, A., Russo, A., and Saponaro, C. 2008. ROME4EU: A Web Service-Based Process-Aware System for Smart Devices. In *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*. Lecture Notes in Computer Science, vol. 5364. Springer, 726–727.

Casati, F., Ceri, S., Paraboschi, S., and Pozzi, G. 1999. Specification and Implementation of Exceptions in Workflow Management Systems. *ACM Trans. Database Systems 24,* 3, 405–451.

Casati, F. and Shan, M. 2001. Dynamic and Adaptive Composition of e-Services. *Information Systems 26,* 3, 143–163.

Catarci, T., de Leoni, M., De Rosa, F., Mecella, M., Poggi, A., Dustdar, S., Juszczyk, L., Truong, H., and Vetere, G. 2007. The WORKPAD P2P Service-Oriented Infrastracture for Emergency Management. In *WETICE '07: Proceedings of the 16th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE, Washington, DC, USA.

Catarci, T., de Leoni, M., Marrella, A., Mecella, M., Vetere, G., Salvatore, B., Dustdar, S., Juszczyk, L., Manzoor, A., and Truong, H.-L. 2008. Pervasive Software Environments for Supporting Disaster Responses. *IEEE Internet Computing 12,* 1, 26–37.

Catarci, T., De Rosa, F., de Leoni, M., Mecella, M., Angelaccio, M., Dustdar, S., Krek, A., Vetere, G., Zalis, Z. M., Gonzalvez, B., and Iiritano, G. 2006. WORKPAD: 2-Layered Peer-to-Peer for Emergency Management through Adaptive Processes. In *CollaborateCom 2006: Proceedings of the 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing*. IEEE.

Cetina, C., Giner, P., Fons, J., and Pelechano, V. 2009. Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. *IEEE Computer 42,* 10, 37–43.

Chiu, D., Li, Q., , and Karlapalem, K. 2000. A logical framework for exception handling in ADOME workflow management system. In *CAiSE2000: Proceedings of 12th International Conference Advanced Information Systems Engineering*. Lecture Notes in Computer Science, vol. 1789. Springer, 110–125.

Cosa GmbH. 2008. COSA BPM product description. `http://www.cosa.de/project/docs/en/COSA57-Productdescription.pdf`. Prompted on 1 February, 2009.

De Giacomo, G., Lespérance, Y., Levesque, H. J., and Sardina, S. 2009. IndiGolog: A high-level programming language for embedded reasoning agents. In *Multi-Agent Programming: Languages, Platforms and Applications*, R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, Eds. Springer, New York, USA, Chapter 2, 31–72. ISBN: 978-0-387-89298-6.

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

de Leoni, M. 2009. Adaptive Process Management in Highly Dynamic and Pervasive Scenarios. In *YR-SOC 2009: Proceedings Fourth European Young Researchers Workshop on Service Oriented Computing*. Electronic Proceedings in Theoretical Computer Science (EPTCS), vol. 2. arXiv.org, 83–97.

de Leoni, M., De Giacomo, G., Lespérance, Y., and Mecella, M. 2009. On-line Adaptation of Sequential Mobile Processes Running Concurrently. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 1345–1352.

de Leoni, M., De Rosa, F., Marrella, A., Poggi, A., Krek, A., and Manti, F. 2007. Emergency Management: from User Requirements to a Flexible P2P Architecture. In *Proceedings of the 4th International Conference on Information Systems for Crisis Response and Management ISCRAM2007*, B. Van de Walle, P. Burghardt, and C. Nieuwenhuis, Eds.

de Leoni, M., Marrella, A., Mecella, M., Valentini, S., and Sardina, S. 2008. Coordinating Mobile Actors in Pervasive and Mobile Scenarios: An AI-based Approach. In *WETICE'08: Proceedings of the 17th IEE International Workshops on Enabling Technologies: Infrastructure for collaboration enterprises*. IEEE, 82–88.

de Leoni, M., Mecella, M., and De Giacomo, G. 2007. Highly Dynamic Adaptation in Process Management Systems Through Execution Monitoring. In *BPM'07: Proceedings of the 5th Internation Conference on Business Process Management*. Lecture Notes in Computer Science, vol. 4714. Springer, 182–197.

Du, W., Davis, J., and Shan, M.-C. 1997. Flexible specification of workflow compensation scopes. In *GROUP '97: Proceedings of the international ACM SIGGROUP conference on Supporting group work*. ACM, New York, NY, USA, 309–316.

Eder, J. and Liebhart, W. 1995. The Workflow Activity Model WAMO. In *COOPIS 1995:Proceedings of the 3rd international conference on Cooperative Information Systems*. Springer, 87–98.

Eder, J. and Liebhart, W. 1996. Workflow recovery. In *COOPIS '96: Proceedings of the First IFCIS International Conference on Cooperative Information Systems*. IEEE Computer Society, Washington, DC, USA, 124.

Ellis, C. and Keddara, K. 2000. A Workflow Change is a Workflow. In *Business Process Management. Models, Techniques and Empirical Studies*. LNCS 1806.

Franke, J., Charoy, F., Ulmer, C., and Antipolis, S. 2010. A Model for Temporal Coordination of Disaster Response Activities. In *ISCRAM 2010: Proceedings of the 4th International Conference on Information Systems for Crisis Response and Management*, C. Z. Simon Trench, Brian Tomaszewski, Ed.

Garlan, D., Kramer, J., and Wolf, A., Eds. 2002. *Proceedings of the First Workshop on Self-Healing Systems (WOSS 2002)*.

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

Golani, M. and Gal, A. 2005. Flexible business process management using forward stepping and alternative paths. In *BPM 2005: Proceedings of the 3rd Internation Conference on Business Process Management*. 48–63.

Göser, K., Jurisch, M., Acker, H., Kreher, U., Lauer, M., Rinderle, S., Reichert, M., and Dadam, P. 2007. Next-generation Process Management with ADEPT2. In *Proceedings of the BPM Demonstration Program at the Fifth International Conference on Business Process Management (BPM'07)*. CEUR Workshop Proceedings, vol. 272. CEUR-WS.org.

Hagen, C. and Alonso, G. 2000. Exception Handling in Workflow Management Systems. *IEEE Trans. Software Engineering 26,* 10, 943–958.

Heimann, P., Joeris, G., Krapp, C., and Westfechte, B. 1996. DYNAMITE: Dynamic Task Nets for Software Process Management. In *Proc. ICSE 1996*.

Humayoun, S. R., Catarci, T., de Leoni, M., Marrella, A., Mecella, M., Bortenschlager, M., and Steinmann, R. 2009. Designing Mobile Systems in Highly Dynamic Scenarios. The WORKPAD Methodology. *Journal on Knowledge, Technology & Policy 22,* 1, 25–43.

Humayoun, S. R., Catarci, T., Leoni, M., Marrella, A., Mecella, M., Bortenschlager, M., and Steinmann, R. 2009. The WORKPAD User Interface and Methodology: Developing Smart and Effective Mobile Applications for Emergency Operators. In *UAHCI '09: Proceedings of the 5th International Conference on Universal Access in Human-Computer Interaction. Part III*. Lecture Notes in Computer Science, vol. 5616. Springer, 343–352.

IBM Inc. 2008. An introduction to WebSphere Process Server and WebSphere Integration Developer. `ftp://ftp.software.ibm.com/software/integration/wps/library/WSW14021-USEN-01.pdf`. Prompted on 1 February, 2009.

Jaccheri, M. and Conradi, R. 1993. Techniques for Process Model Evolution in EPOS. *IEEE Trans. Software Engineering 19,* 12.

Kinateder, M. 2006. Sap advanced workflow techniques. `https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/docs/library/uuid/82d03e23-0a01-0010-b482-dccfe1c877c4`. Prompted on 1 February, 2009.

Kowalski, R. A. 1995. Using meta-logic to reconcile reactive with rational agents. *Meta-logics and logic programming*, 227–242.

Ly, L. T., Rinderle, S., and Dadam, P. 2008. Integration and verification of semantic constraints in adaptive process management systems. *Data & Knowledge Engineering 64,* 1, 3–23.

McCarthy, J. and Hayes, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence 4*, 463–502.

Mecella, M., de Leoni, M., Marrella, A., Catarci, T., Bortenschlager, M., and Steinmann, R. 2010. The WORKPAD Project Experience: Improving the Disaster Response through Process Management and Geo Collaboration. In *Proceedings of the 7th International Conference on Information Systems for Crisis Response and Management (ISCRAM2010).*

Milner, R. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, vol. 92. Springer.

Müller, R., Greiner, U., and Rahm, E. 2004. AGENTWORK: a workflow system supporting rule-based workflow adaptation. *Data & Knowledge Engineering 51,* 2, 223–256.

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

Rinderle, S., Reichert, M., and Dadam, P. 2004. Correctness Criteria for Dynamic Changes in Workflow Systems - A Survey. *Data & Knowledge Engineering 50*.

Sadiq, S. W., Marjanovic, O., and Orlowska, M. E. 2000. Managing Change and Time in Dynamic Workflow Processes. *International Journal of Cooperative Information Systems 9,* 1–2, 93–116.

ter Hofstede, A., van der Aalst, W., Adams, M., and Russell, N. 2010. *Modern Business Process Automation: YAWL and its Support Environment*. Springer.

Tibco Software Inc. 2008. Introduction to TIBCO iProcess Suite. `www.tibco.com/resources/software/bpm/tibco\_iprocess\_suite\_whitepaper.pdf`. Prompted on 1 February, 2009.

van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. 2003a. Workflow Patterns. *Distributed and Parallel Databases 14,* 1, 5–51.

van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. 2003b. Workflow Patterns. *Distributed and Parallel Databases 14,* 1, 5–51.

Wagenknecht, A. and Rüppel, U. 2009. Improving Resource Management In Flood Response With Process Models and Web GIS. In *Proceedings of the 16th TIEMS Annual Conference*. International Emergency Management Society.

Weber, B., Reichert, M., Rinderle-Ma, S., and Wild, W. 2009. Providing Integrated Life Cycle Support in Process-Aware Information Systems. *International Journal of Cooperative Information Systems (IJCIS) 18,* 1, 115–165.

Weske, M. 2001. Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System. In *HICSS01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. IEEE Computer Society.

Weske, M. 2007. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag Berlin Heidelberg.

APPENDIX

A. PRELIMINARIES

In this appendix, we provide a brief overview of the logical framework used to formalize SmartPM and its adaptation features. The *Situation Calculus* [McCarthy and Hayes 1969; Reiter 2001] is a logical language specifically designed for representing dynamically changing worlds in which all changes are the result of named *actions*. A possible history of actions is represented by a so-called *situation*, a first-order term in the language. The constant S_0 denotes the initial situation, where no actions have yet been performed. Sequences of actions are built using the special function symbol *do*: $do(a, s)$ denotes the successor situation resulting from performing action a in situation s. Properties that hold in a situation are called *fluents*. Technically, these are predicates taking a situation term as their last argument. For example, fluent $DoorOpen(x, s)$ may denote that door x is open in situation s. A distinguished predicate $Poss(a, s)$ is used to state that action a is executable in s, i.e., the precondition of action a.

Within this language, one can formulate *action theories* describing how the world changes as the result of the available actions. For instance, a *basic action theory* [Reiter 2001] is built from a set of domain-independent foundational axioms to describe the structure of situations, one successor state axiom per fluent (capturing the effects and non-effects of actions), one precondition axiom per action (specifying when the action is executable), and initial state axioms describing what is true initially (i.e., what is true in the initial situation S_0). For example, the successor state axiom for fluent $DoorOpen(x, s)$ and the precondition for action *open* are as follows:

$$\begin{aligned} &DoorOpen(x, do(a, s)) \equiv \\ &\quad (a = open(x) \wedge \neg Locked(x, s)) \vee DoorOpen(x, s) \wedge a \neq close(x); \\ &Poss(open(x), s) \equiv \neg DoorOpen(x, s). \end{aligned}$$

That is, a door is open after an action a has been performed iff a denotes the action of opening that door and the door is not locked, or the door was open before a and a action is not that one of closing it. The action of opening a door is possible if the door is closed.

On top of Situation Calculus action theories, logic-based programming languages can be defined, which, in addition to the primitive actions, allow the definition of *complex* actions. The IndiGolog language [De Giacomo et al. 2009] is a high-level programming language equipped with standard imperative constructs (e.g., sequence, conditional, iteration, etc) as well as procedures and primitives for expressing various types of concurrency and prioritized interrupts. More interesting, the language includes *nondeterministic constructs* to accommodate loose specification of programs by allowing "*gaps*" that ought to be resolved by the reasoner/-planner or executor. To resolve such gaps successfully at execution time, IndiGolog includes a lookahead operator. The complete set of constructs available in IndiGolog is summarized in Table III.

The interrupt construct is also of particular interest. Regarding interrupts, it

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
$proc\ P(\vec{v})\ \delta$	Invocation of a procedure passing a vector $\vec{v}$ of parameters
$(\delta_1 \vert \delta_2)$	Non-deterministic choice among (sub-)program δ_1 and δ_2.
$if\ \phi\ then\ \delta_1\ else\ \delta_2$	Conditional statement: if ϕ holds, δ_1 is executed; otherwise δ_2.
$while\ \phi\ do\ \delta$	Iterative invocation of δ
$(\delta_1 \parallel \delta_2)$	Concurrent execution
δ^*	Non-deterministic iteration of program execution
$\Sigma(\delta)$	Emulating off-line execution
$\pi a.\delta$	Non-deterministic choice of argument a followed by the execution of δ.
$\langle \phi \rightarrow \delta \rangle$	δ is repeatedly executed until ϕ becomes false, releasing control to anyone else able to execute.

Table III. IndiGolog constructs

turns out that these can be explained using other constructs of IndiGolog

$$\langle\ \phi \rightarrow \delta\ \rangle \stackrel{\text{def}}{=} \textbf{while}\ Interrupts\_running\ \textbf{do}$$
$$\textbf{if}\ \phi\ \textbf{then}\ \delta\ \textbf{else}\ \texttt{false}\ \textbf{endIf}$$
$$\textbf{endWhile}$$

To see how this works, first assume that the special fluent *Interrupts_running* is identically `true`. When an interrupt $\langle \phi \rightarrow \delta \rangle$ gets control, it repeatedly executes δ until ϕ becomes false, at which point it blocks, releasing control to anyone else able to execute. The control release also occurs if ϕ cannot progress (e.g., since no action meets its precondition).

From the formal point of view, IndiGolog programs are just logical *terms* and the language (single-step) semantics is expressed using the following two predicates:

—$Trans(\delta', s', \delta'', s'')$: a single step of program δ' in situation s' may lead to situation s'' with δ'' remaining to be executed.

—$Final(\delta', s')$: program δ' may legally terminate in situation s'.

With these two predicates characterized via a set of axioms, it is not difficult to define what is means to execute a program. Originally, in languages like Golog and ConGolog, programs were conceived to be executed (verified) *offline*; the interpreter looks for a sequence of actions $[a_1, \ldots, a_m]$ such that $Do(\delta, s, do(a_m, do(a_{m-1}, \ldots, do(a_1, S_0))))$ is entailed by the specification, where $Do(\delta, s, s')$ is intended to mean that situation s' represents a legal execution of program δ starting from situation s; formally:

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

where $Trans^*$ stands for the reflexive transitive closure of *Trans*.

Clearly, this type of execution remains infeasible for large programs and precludes both runtime sensing information and reactive behavior. To deal with these drawbacks, IndiGolog provides a formal notion of *interleaved planning, sensing, and action*. Roughly speaking, an *online execution* of program finds a next possible action, executes it in the real world, obtains sensing information afterward, and repeats the cycle until the program is finished. The fact that actions are quickly executed without much deliberation and sensing information is gathered after each step makes the approach realistic for dynamic and changing environments. To cope

with nondeterministic choices in programs and the impossibility of backtracking actions in the real world, IndiGolog provides a planning construct $\Sigma(\delta)$—-the *search operator*—as a local, controlled form of off-line verification such that the amount of lookahead to be performed is under the control of the programmer. When the executor reaches a program of the form $\Sigma(\delta)$, a complete successful execution of program δ is searched before even the first action is performed. This search, of course, implies reasoning about the various different executions that δ may yield. For more details on IndiGolog, we refer the reader to [De Giacomo et al. 2009].

We close the appendix by noting that, besides the solid theoretical foundation behind IndiGolog, a full-fledged sophisticated interpreter for it is freely available.[6]

B. TRANSLATION FROM BPEL4WS TO INDIGOLOG AND SITUATION CALCULUS

Process designers are not intended, in our approach, to define processes using directly the SmartPM framework. Conversely, on the basis of the requirements stemming from the WORKPAD project, we assume they specify the processes in WS-BPEL, and have available a kind of templates (specific for different scenarios) for specifying "semantic aspects" needed in our approach (e.g., pre- e post-conditions, etc.). In the future, we aim to consider other (conceptual) languages for process specifications, in particular BPMN – Business Process Modeling Notation – and YAWL.

Therefore, we provide in SmartPM a module able to translate (most of) WS-BPEL specifications into IndiGolog programs and complaint Situation Calculus domain theories[7]. There are valuable features of IndiGolog for processes modeling that WS-BPEL is unable to model. Firstly, designers are able to formalize explicitly the pre-conditions of tasks. Secondly, SmartPM is able to describe conditions which do not model WS-BPEL-variables (or equivalent). Unlike WS-BPEL, IndiGolog can accommodate the specification and dynamics of resources (e.g., the capabilities of human beings and services) that can be of some interest during the process performance. More generally, in IndiGolog one can specify other properties of the world that can affect the process execution. All these aspects, which are not directly expressible in WS-BPEL, are added in our approach as XML annotations to the WS-BPEL specification. Though not mandatory, they may accommodate extra information about the process execution environment; this information may allow for more accurate sensing of deviations and, hence, yield better process adaptation. In our validation, we have provided such annotations as pre-defined templates, in order to ease as much as possible the specification by the process designers.

We provide here some insight into the technique for translating WS-BPEL specifications into IndiGolog. On the basis of such a technique, SmartPM supports at

[6] `http://sourceforge.net/projects/indigolog/`

[7] A feature we do not consider is one-way jumps. This does not limit the construct supported, as indeed, no workflow patterns [van der Aalst et al. 2003a] have been implemented in [ter Hofstede et al. 2010] through one-way jumps. Indeed, one-way jumps, which are similar to arbitrary GOTO statements, are often unnecessary, since one can transform a process specification to one containing no one-way jumps, while the same behavior is observed.

BPEL	Sample Code	Translation

assign	`<assign name="A">}`
	`<copy>`
	`<from variable="source">`
	`<to variable="dest">`
	`</copy>`
	`</assign>`
		PROC **A**()
		1 $Copy_{source}^{dest}$;

invoke	`<invoke name="B"`
	`inputVar="input"`
	`outputVar="output"`
	`operation= "task1" />`
		PROC **B**()
		1 **ManageExecution**($[Task, input]$);

sequence	`<sequence name="Seq">`
	`<...subProc1...>`
	`. . .`
	`<...subProcN...>`
	`</sequence>`
		PROC **Seq**()
		1 (**subProc1**(); ...; **subProcN**());

flow	`<flow name="Parallel">`
	`<...subProc1...>`
	`. . .`
	`<...subProcN...>`
	`</flow>`
		PROC **Parallel**()
		1 (**subProc1**() ‖ ... ‖ **subProcN**());

if	`<if name="Select">`
	`<condition>cond</condition>`
	`<...subProc1...>`
	`<else>`
	`<...subProc2...>`
	`</else>`
	`</if>`
		PROC **Select**()
		1 **if** *cond*1
		2 **then subProc1**()
		3 **else subProc2**()

while	`<while name="Loops"`
	`condition="cond">`
	`<...subProc1...>`
	`</while>`
		PROC **Loops**()
		1 **while** *cond*
		2 **do subProc1**()

for each	`<forEach name="FE" parallel="yes"`
	`countername="n">`
	`<startCounterVal>MinVal</startCounterVal>`
	`<finalCounterVal>MaxVal</finalCounterVal>`
	`<scope>`
	`<variables>`
	`<variable name="XYZ" type="ABC" />`
	`...`
	`</variables>`
	`< ... name="branch" ...>`
	`</scope>`
	`</forEach>`
		PROC **FE**()
		1 **FE_Helper**($condMinVal$)
		
		PROC **FE_Helper**(n)
		1 **if** ($n \leq condMaxVal$)
		2 **then FE_Helper**($n+1$) ‖ **branch**(n)

Table IV. From WS-BPEL to IndiGolog

least the same workflow patterns as WS-BPEL does. A comprehensive analysis of the patterns supported by WS-BPEL is illustrated in [ter Hofstede et al. 2010].

We start by describing the syntax of variable declarations. In order to allow for complex data representation when designing a process, each variable is represented as an XML complex type, whose definition is provided with a further XML schema.

```
<variables>
  <variable name="A" type="xs:A-type" >
  . . . . .
  <variable name="X" type="xs:X-type" >
```

```
</variables>
```

When mapping to the SmartPM model, each variable defined in the WS-BPEL specification is translated to a Situation Calculus fluent of the same name (e.g., *A(s)*, ..., *X(s)* above). The values of such fluents describe the state of the process instance in a specific situation. In general, each fluent can assume values that are denoted by complex tuples. Together with the fluent, the translation also generates a situation independent predicate (e.g., *DomainA*, ... , *DomainX* above) that states the admissible values for a given fluent, i.e., its domain. The generic successor-state axiom is as follows:

$$\begin{array}{l} X(do(a,s)) = y \equiv \\ \quad [\exists srvc.a = \mathit{finishedTask}(srvc, \bar{t}, V) \wedge V = \bar{i} \wedge \mathit{DomainX}(y)] \bigvee \\ \quad [X(s) = y \wedge \\ \quad \neg\exists srvc.a = \mathit{finishedTask}(srvc, \bar{t}, V') \wedge V' = \bar{i}] \end{array} \tag{7}$$

where $\bar{t}$ and $\bar{i}$ are, respectively, the task and the relevant input which can modify the value of the variable/fluent X.

Table IV shows how the typical WS-BPEL's constructs can be translated in IndiGolog and Situation Calculus. Each construct is mapped onto an IndiGolog procedure. With the exception of primitive constructs for variable assignment and task invocation, the structured constructs can be nested arbitrarily. Consequently, IndiGolog procedures may call, in turn, other procedures.

We focus now on the `for-each` construct, which structure is shown in the lower part of Table IV. It allows to support a special kind of forks known in the literature as "Multiple Instances with a Priori Run-Time Knowledge" [van der Aalst et al. 2003b] (a.k.a. "multitask"), which is very important in concrete scenarios. The number of branches that are initiated depends on some variable values. Once initiated, these branches are independent of each other and run concurrently. Every branch can define its own variables whose scope is local of the specific variable. It follows that there exist multiple variables of the same name, once per branch. The reader should note note the recursive invocation of procedure **FE_Helper** that takes the counter value as input. At each recursive step, a new invocation of procedure **branch**(n) is done, and, concurrently, **FE_Helper** is invoked again after incrementing the counter by 1. Procedure **branch**(n) maps the generic branch and its definition follows the guidelines as from Table IV. When mapping the generic branch, the only difference is that it needs to take the current counter value as input. The same would also hold for the possible nested procedures. The current counter value is, then, required to be passed to every invocation of **ManageExecution**($[Task, (input, n)]$). The counter value is also required to be returned to SmartPM in the output set when services perform $\mathit{finishedTask}(srvc, t, (output, n))$.

When variables are defined in the scope of a `for-each` branch, they are local. IndiGolog does not support local fluents, i.e. fluents accessible only inside given procedures. Since fluents are always global, we need to refine the notion of mapping of variables to fluents with respect to the one defined above.

For the sake of explanation, let us assume that the WS-BPEL specification defines a variable `Z` in the scope of a `for-each` construct. This is mapped into a functional

fluent $Z(n, s)$ where the first argument n maps the local WS-BPEL's variable of n-th branch. In the light of above, when fluents map local variables of WS-BPEL `for-each` branches, the following is a refinement of the generic successor state axiom given in Equation 7:

$$\begin{array}{l} Z(n, do(a, s)) = y \equiv \\ \quad [\exists srvc.a = \mathit{finishedTask}(srvc, \bar{t}, (V, n)) \wedge V = \bar{i} \wedge \mathit{DomainZ}(y)] \bigvee \\ \quad [Z(n, s) = y \wedge \\ \quad \neg\exists srvc.a = \mathit{finishedTask}(srvc, \bar{t}, (V', n)) \wedge V' = \bar{i}] \end{array} \tag{8}$$

where $\bar{t}$ and $\bar{i}$ are, respectively, the task and the relevant input which can modify the value of the variable/fluent Z.

The translation of the constructs *wait*, *receive*, *reply* and *repeat-until* can be easily derived from those which we have already discussed.The complete WS-BPEL code of the process shown in Section 6 and its translation in Situation Calculus and in the executable IndiGolog code can be downloaded at `http://www.dis.uniroma1.it/~marrella/public/TAAS/appendixes.zip`