SmartPM: An Adaptive Process Management System through Situation Calculus, IndiGolog, and Classical Planning

Andrea Marrella and **Massimo Mecella**
Sapienza - Universitá di Roma, Rome, Italy
{marrella|mecella}@dis.uniroma1.it

Sebastian Sardina
RMIT University, Melbourne, Australia
sebastian.sardina@rmit.edu.au

Abstract

In this paper we present SmartPM, a model and a prototype Process Management System featuring a set of techniques providing support for automated adaptation of knowledge-intensive processes at run-time. Such techniques are able to automatically adapt process instances without explicitly defining policies to recover from exceptions and without the intervention of domain experts at run-time, aiming at reducing error-prone and costly manual ad-hoc changes, and thus at relieving users from complex adaptations tasks. To accomplish this, we make use of well-established techniques and frameworks from Artificial Intelligence, such as situation calculus, IndiGolog and classical planning.

1 Introduction

This paper is concerned with the use of of three well-established KR&R techniques—reasoning about actions, high-level programming, and automated planning—for the *automated adaptation* of knowledge-intensive processes that execute in highly dynamic settings. The work falls within the scope of *Business Process Management* (BPM) (van der Aalst, ter Hofstede, and Weske 2003), an active area of research that is highly relevant from a practical point of view while offering many technical challenges.

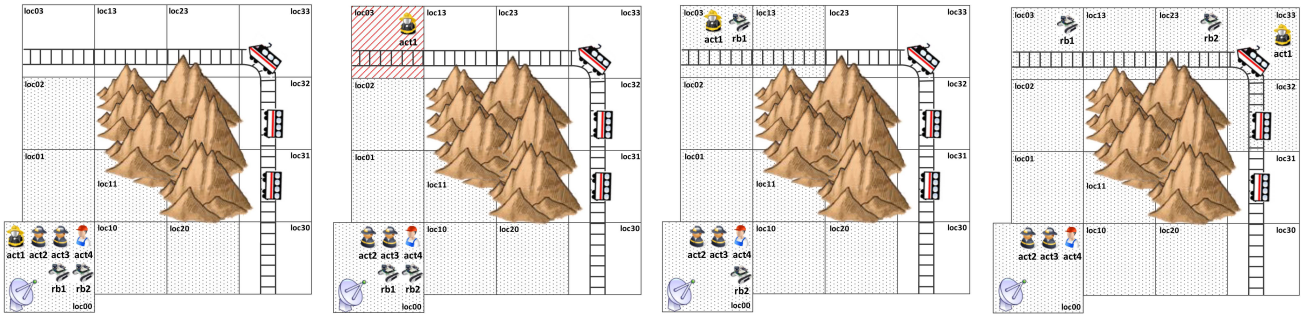
BPM is based on the observation that each product and/or service that a company provides to the market is the outcome of a number of activities performed. *Business processes* are the key instruments for organizing such activities and improving the understanding of their interrelationships. Examples of traditional business processes include insurance claim processing, order handling, and personnel recruitment. In order to support the design and automation of business processes, a new generation of information systems, called *Process Management Systems* (PMSs) have become increasingly popular during the last decade (Weske 2012). A PMS is a software system that manages and executes business processes involving people, applications, and information sources on the basis of *process models* (Dumas, van der Aalst, and ter Hofstede 2005). The basic constituents of a process model are *tasks*, describing the various activities to be performed by process participants (i.e., software applications, agents, or humans). The procedural rules to control such tasks, described by so-called "routing" constructs such as sequences, loops, parallel and alternative branches, define the *control flow* of the process. A PMS, then, takes a process model (containing the process' tasks and control flow) and manages the process routing by deciding which tasks are enabled for execution. Once a task is ready for execution, the PMS assigns it to those participants capable of carrying it on.

Current maturity of process management methodologies has led to the application of process-oriented approaches in new rich challenging scenarios, such as healthcare (Lenz and Reichert 2007), emergency management (Marrella, Russo, and Mecella 2012), and home automation (Helal et al. 2005). In those settings, processes generally reflect "preferred work practices", and the control flow is influenced by user decision making and coupled with contextual data and knowledge production. Such processes are known as *knowledge-intensive* processes (KiPs) (Di Ciccio, Marrella, and Russo 2012)—genuinely knowledge and data centric—and require the integration of the data dimension with the traditional control flow dimension.

During the enactment of KiPs, variations or divergence from structured reference models are common due to exceptional circumstances arising (e.g., autonomous user decisions or contextual changes), thus requiring the ability to properly *adapt* the process behavior (Sadiq, Sadiq, and Orlowska 2001). In knowledge-intensive scenarios, the fact is that the number of possible exceptions is often too large, and traditional manual implementation of exception handlers at design-time is not feasible (Reichert and Weber 2012). In fact, the designer often lacks the required knowledge to model all the possible exceptions at the outset.

To tackle this issue, we develop in this paper an approach, together with an actual implementation, to *automatically adapt KiPs at run-time* when unanticipated exceptions occur, thus requiring no specification of recovery policies at design-time. To that end, we shall resort to three popular Artificial Intelligence (AI) "technologies": situation calculus (Reiter 2001), IndiGolog (De Giacomo et al. 2009), and classical planning (Nau, Ghallab, and Traverso 2004; Geffner and Bonet 2013). We use the situation calculus formalism to model the domain in which KiPs are to be executed, including available tasks, contextual properties, tasks'

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.



(a) Initial configuration. (b) Actor *act1* gets to *loc03*. (c) Robot *rb1* re-establishes network connectivity to actor *act1*. (d) Robot *rb2* gets to *loc23* so that *act1* is connected once reached the destination.

Figure 1: A train derailment situation; area and context of the intervention.

preconditions and effects, and the initial state. On top of such a logic-based model, we use the IndiGolog high-level agent programming language for the specification of the structure and control flow of KiPs. We customize IndiGolog to monitor the online execution of KiPs and detect potential mismatches between the model and the actual execution. If an exception invalidates the enactment of the KiP being executed, an external state-of-the-art planner is invoked to synthesise a recovery procedure for adapting the faulty process instance. We refer to this adaptive framework for KiPs as SmartPM (Smart Process Management), described in Section 4.

Besides providing the conceptual framework, we validated the approach with a case study based on real KiPs coming from an emergency management domain (Section 5). To do so, we have implemented SmartPM by relying on an existing IndiGolog interpreter (De Giacomo et al. 2009) and the state-of-the-art planning system LPG-td (Gerevini, Saetti, and Serina 2003). In Section 6, we compare SmartPM with other works that exploit AI techniques for enhanced process adaptation. We conclude, in Section 7, by drawing conclusions and discussing limitations and future work.

Before jumping to the technical proposal, let us first start with an overview of our case study and some preliminary notions necessary to understand the rest of the paper.

2 Case Study

Our case study and evaluation involves a disaster management inspired by the WORKPAD project.[1] In particular, it concerns the emergency management scenario described in Figure 1(a), in which a train derailment situation is depicted in a grid-type map. For the sake of simplicity, the train is composed of a locomotive (located at *loc33*) and two coaches (located at *loc32* and *loc31*, resp.).

During the management of complex emergency scenarios, teams of *first responders* act in disaster locations to achieve specific goals. In our train derailment situation, the goal of an incident response plan is to evacuate people from the coaches and take pictures for evaluating possible damages to the locomotive. To that end, a response team is sent to the derailment scene. The team is composed of four first responders, called *actors*, and two *robots*, initially all located at location cell *loc00*. It is assumed that actors are equipped with mobile devices for picking up and executing tasks, and that each provide specific capabilities. For example, actor *act1* is able to extinguish fire and take pictures, while *act2* and *act3* can evacuate people from train coaches. The two robots, in turn, are designed to remove debris from specific locations. When the battery of a robot is discharged, actor *act4* can charge it.

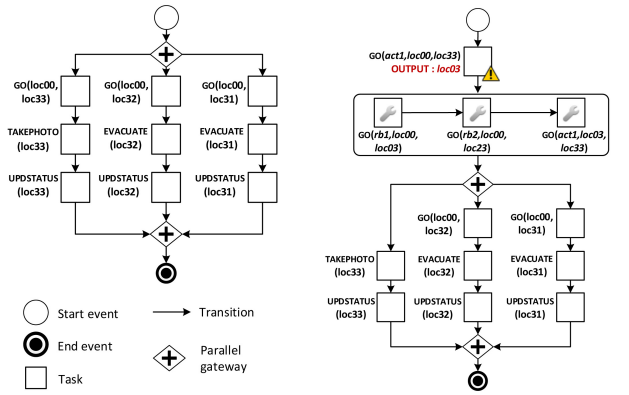
In order to carry on the response plan, all actors and robots ought to be continually inter-connected. The connection between mobile devices is supported by a fixed antenna located at *loc00*, whose range is limited to the dotted squares in Figure 1(a). Such a coverage can be extended by robots *rb1* and *rb2*, which have their own independent (from antenna) connectivity to the network and can act as wireless routers to provide network connection in all adjacent locations.

An incident response plan is defined by a set of activities that are meant to be executed on the field by first responders, and are predicated on specific contexts. Therefore, the information collected on-the-fly is used for defining and configuring at run-time the incident response plan at hand. A possible concrete realization of an incident response plan for our scenario is shown in Figure 2(a), using the Business Process Model and Notation (BPMN).[2] The process is composed of three parallel branches with tasks instructing first responders to act for evacuating people from train coaches, taking pictures of the locomotive, and assessing the gravity of the accident. Due to the high dynamism of the environment, there are a wide range of exceptions that can ensue. Because of that, there is not a clear anticipated correlation between a change in the context and a change in the process.

So, suppose for instance that actor *act1* is sent to the locomotive's location, by assigning to it the task GO(*loc00,loc33*) in the first parallel branch. Unfortunately, however, the actor happens to reach location *loc03* instead. The actor is now located at a different position than the de-

[1] See `http://www.dis.uniroma1.it/~workpad/`

[2] See `www.omg.org/spec/BPMN/`



(a) Main process (b) Adapted process

Figure 2: An emergency response plan and its adaptation.

sired one, and most seriously, is out of the network connectivity range (Figure 1(b)). Since all participants need to be continually inter-connected to execute the process, the PMS has to first find a recovery procedure to bring back full connectivity, and then find a way to re-align the process. To that end, provided robots have enough battery charge, the PMS may first instruct the first robot to move to cell *loc03* (Figure 1(c)) in order to re-establish network connection to actor *act1*, and then instruct the second robot to reach location *loc23* in order to extend the network range to cover the locomotive's location *loc33*. Finally, the task GO(*loc03,loc33*) is (re)assigned to actor *act1* (Figure 1(d)). The corresponding updated process is shown in Figure 2(b), with the encircled section being the recovery (adaptation) procedure. Notice that after the recovery procedure, the enactment of the original process can be resumed to its normal flow.

The point is that it is not adequate to assume that the process designer can pre-define all possible recovery activities for dealing with exceptions in domains that are knowledge-intensive as the one just described: the recovery procedure will depend on the actual context (e.g., the positions of participants, the range of the main network, robot's battery levels, whether a location has become dangerous to get it, etc.) and there are too many of them to be considered.

3 Preliminaries

Situation Calculus The *situation calculus* is a logical language designed for representing and reasoning about dynamic domains (Reiter 2001). The dynamic world is seen as progressing through a series of situations as a result of various *actions* being performed. A *situation* s is a first-order term denoting the sequence of actions performed so far. The special constant S_0 stands for the initial situation, where no action has yet occurred, whereas a special binary function symbol $do(a, s)$ denotes the situation resulting from the performance of action a in situation s. Features of the world whose truth value may change from situation to situation are modeled by means of so-called *fluents*. Technically, fluents are predicates taking a situation term as their last argument. For example, fluent $Holding(x, y, s)$ may state that entity x is holding object y at situation s. A special predicate $Poss(a, s)$ is used to state that action a is executable in situation s. We write $\phi(\vec{x})$ to denote a formula whose free variables are among variables $\vec{x}$. A fluent-formula is one whose only situation term mentioned is situation variable s.

Within this language, one can formulate action theories describing how the world changes as the result of the available actions. A *basic action theory* (BAT) (Reiter 2001) $\mathcal{D} = \Sigma \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{poss} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una}$ includes domain-independent foundational axioms to describe the structure of situations (Σ), one successor state axiom per fluent capturing the effects and non-effects of actions ($\mathcal{D}_{ss}$), one precondition axiom per action specifying when the action is executable ($\mathcal{D}_{poss}$), unique name axioms for actions ($\mathcal{D}_{una}$) and initial state axioms describing what is true initially in S_0 ($\mathcal{D}_{S_0}$). In particular, the *successor state axiom* for a fluent $F(\vec{x}, s)$ is an axiom of the form $F(\vec{x}, do(a, s)) \equiv \Psi_F(\vec{x}, a, s)$,[3] where $\Psi_F(\vec{x}, a, s)$ is a fluent-formula characterizing the dynamics of fluent $F(\vec{x}, s)$. Importantly, $\Psi_F(\vec{x}, a, s)$ can accommodate Reiter (2001)'s solution to the frame problem. In addition, precondition axioms are of the form $Poss(a(\vec{x}), s) \equiv \Pi_a(\vec{x}, s)$, where $\Pi_a(\vec{x}, s)$ is a fluent-formula defining the conditions under which action a can be legally executed in situation s. Finally, $\mathcal{D}_{S_0}$ is a collection of first-order sentences whose only situation term mentioned is S_0.

IndiGolog 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. In particular, we focus on IndiGolog (De Giacomo et al. 2009), the latest in the Golog-like family of programming languages for autonomous agents providing a formal account of interleaved action, sensing, and planning. IndiGolog programs are meant to be executed *online*, in that, at every step, a legal next action is selected for execution, performed in the world, and its sensing outcome gathered. To account for planning, a special look-ahead construct $\Sigma(\delta)$—the search operator—is provided to encode the need for solving (i.e., finding a complete execution) program δ offline.

IndiGolog allows us to define every well-structured process as defined in (van der Aalst et al. 2003); it is equipped with all standard imperative constructs (e.g., sequence, conditional, iteration, etc.) to be used on top of situation calculus primitives actions. An IndiGolog program is meant to run relative to an action theory, providing meaning to primitive actions and conditions in the program. Here we concentrate on the fragment defined by the following constructs:

a	atomic action
$\phi?$	test for a condition
$\delta_1; \delta_2$	sequence
$\pi x.\delta(x)$	nondeterministic choice of argument
δ^*	nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2 **endIf**	conditional
while ϕ **do** δ **endWhile**	while loop
proc $P(\vec{x})$ **do** $\delta(x)$ **endProc**	procedure

[3] Free variables are assumed to be universally quantified.

$\delta_1 \| \delta_2$	concurrency
$\delta_1 \rangle\!\rangle \delta_2$	prioritized concurrency
$\langle \phi \rightarrow \delta \rangle$	interrupt
$\Sigma(\delta)$	lookahead search

Test program $\phi?$ can be executed if condition ϕ holds true, whereas program $\pi x.\delta(x)$ executes program $\delta(x)$ for *some* nondeterministic choice of a binding for variable x, and δ^* executes δ zero, one, or more times. The interleaved concurrent execution of two programs is represented with constructs $\delta_1 \| \delta_2$ and $\delta_1 \rangle\!\rangle \delta_2$; the latter considering δ_1 at higher priority level (i.e., δ_2 can perform a step only if δ_1 is blocked or completed). Finally, interrupt $\langle \phi \rightarrow \delta \rangle$ states that program δ ought to be executed to completion if ϕ happens to become true, whereas $\Sigma(\delta)$ finds and executes a plan—a sequence of actions—that is guaranteed to fully execute δ.

By properly combining prioritized concurrency and interrupts, together with IndiGolog's default online execution style, it is possible to design processes that are sufficiently open and reactive to dynamic environments. Furthermore, by resorting to the search operator, one can specify local places in programs where lookahead reasoning is required. Both aspects will end up being fundamental for our adaptive process management framework in the next section.

Classical Planning Planning systems are problem-solving algorithms that operate on explicit representations of states and actions (Nau, Ghallab, and Traverso 2004; Geffner and Bonet 2013). PDDL (Edelkamp and Hoffmann 2004) is the standard planning representation language; it allows one to formulate a *planning problem* $\mathcal{P} = \langle I, G, \mathcal{P}_D \rangle$, where I is the initial state, G is the goal state, and $\mathcal{P}_D$ is the planning domain. In turn, a planning domain $\mathcal{P}_D$ is built from a set of *propositions* describing the state of the world (a state is characterized by the set of propositions that are true) and a set of *operators* (i.e., actions) that can be executed in the domain. Each operator is characterized by its preconditions and effects, stated in terms of the domain propositions.

There exist several forms of planning in the AI literature. In this paper, we focus on *classical planning*, characterized by fully observable, static, and deterministic domains. A solution for a classical planning problem $\mathcal{P}$ is a sequence of operators—a plan—whose execution transforms the initial state I into a state satisfying the goal G. Such a plan is computed in advance and then carried out (unconditionally). The field of classical planning has experimented huge advances in the last twenty years, leading to a variety of concrete solvers (i.e., planning systems) that are able to create plans with thousands of actions for problems containing hundreds of propositions. In this work, we represent planning domains and planning problems using PDDL 2.2 (Edelkamp and Hoffmann 2004), which includes operators with disjunctive preconditions and derived predicates.

4 The SmartPM Approach

We adopt a *service-based* approach to process management. Thus, *tasks are executed by services*, such as software applications, human actors, robots, etc. Each task can be seen as a single step consuming input data and producing output data.

In this section, we show how one can put together the three AI frameworks described above to build a PMS, which we shall name SmartPM, that is able to not only enact KiPs, but also to *automatically* adapt processes in case of unanticipated exceptions. Intuitively, situation calculus theories will be used to model the contextual information in which the process is meant to run, IndiGolog programs will encode the KiP to be carried out, and planning systems will be used to support the automated adaptation of a process when needed.

SmartPM Basic Action Theory

A BAT for a SmartPM application specifies:

1. the tasks and services of the domain of concern;
2. the support framework for managing the task life-cycle;
3. the contextual setting in which processes operate; and
4. the support framework for the monitoring of processes.

Let us start by describing the first three ones. So, to encode tasks and services, we use five non-fluent predicates:

- *Service*(srv): *srv* is a service (i.e., a process participant);
- *Task*(t): t is a task (e.g., GO(l_1, l_2) or TAKEPHOTO(l) may denote the tasks of navigating or taking pictures);
- *Capability*(c): c is a capability;
- *Provides*(srv, c): service *srv* provides capability c; and
- *Requires*(t, c): task t requires the capability c.

Observe all these predicates are rigid: they do not depend on a situation term and are hence static. A service *srv* is able to perform certain task t iff *srv* provides all capabilities required by the task t. This is captured formally using the following abbreviation:

$$\mathit{Capable}(srv, t) \stackrel{\text{def}}{=} \forall c.\mathit{Requires}(t, c) \supset \mathit{Provides}(srv, c).$$

To talk about concrete runs of tasks, we associate them with unique identifiers. A *task instance* is then a tuple $t : id$, where t is a task and *id* is an identifier.

The life-cycle of tasks involves the execution of four primitive actions executed by the PMS and two exogenous actions arising from services. More concretely, the protocol for a successful execution of a task t goes as follows:

1. First, the PMS assigns task instance $t : id$ to a service *srv* by performing primitive action ASSIGN$(srv, id, t, \vec{o_e})$, where $\vec{o_e}$ is the *expected* (sensing result) output.
2. When a service is ready for task execution, it generates the exogenous action READYTOSTART(srv, id, t).
3. Next, the PMS performs primitive action START(srv, id, t) to authorize the service in question that is formally allowed to start carrying out the task instance.
4. When the service completes the task, it generates the exogenous action FINISHED$(srv, id, t, \vec{o_r})$, with $\vec{o_r}$ representing the physical *actual* outcome returned by the task execution (we use ϵ to denote the empty output).
5. At this point, the PMS updates the properties (i.e., the fluents) to reflect the effects of the task just completed.

6. Finally, the PMS acknowledges the completion of the task and releases the service from the task via primitive actions ACKCOMPL(srv, id, t) and RELEASE(srv, id, t).

The intended meaning of all the above actions is captured by means of a set of domain-independent fluents that are used to keep track of the life-cycle of tasks as well as the resource perspective of a process. For example, fluent $\mathit{Free}(srv, s)$ denotes whether a service *srv* is available for task assignments in situation s, whereas $\mathit{ExOut}(id, t, s)$ denotes the expected output of the task; their dynamics are captured via the following successor state axioms (which include Reiter (2001)'s solution to the frame problem):

$$\begin{aligned}&\mathit{Free}(srv, do(a, s)) \equiv\\ &\quad (\exists t, id)a = \text{RELEASE}(srv, id, t) \vee\\ &\quad [\mathit{Free}(srv, s) \wedge (\forall t, id, \vec{o_e})a \neq \text{ASSIGN}(srv, id, t, \vec{o_e})];\\ &\mathit{ExOut}(id, t, do(a, s)) = \vec{o} \equiv\\ &\quad (\exists srv, \vec{o_e})a = \text{ASSIGN}(srv, id, t, \vec{o_e}) \wedge \vec{o} = \vec{o_e} \vee\\ &\qquad \mathit{ExOut}(id, t, s) = \vec{o}.\end{aligned}$$

That is, a service is free (for task assignment) after the execution of an action a iff a releases the service from some task assignment, or it was free before the execution of a and a does not assign a task instance to it. Similarly, the expected output of a task instance is determined by the assignment step (and never changes). Initial expected outcomes are initialized to "no output" via an axiom $(\forall t, id).\mathit{ExOut}(id, t, S_0) = \epsilon$ in $\mathcal{D}_{S_0}$. Other similar fluents are used to manage the life-cycle of tasks.

The BAT shall also contain a set of domain-dependent fluents, together with their corresponding precondition and successor state axioms, capturing the contextual scenario in which the process is meant to to be executed. We call them *data fluents*. In general, such fluents will be affected upon the release of an assignment task, that is, whenever a task is considered fully executed. In addition, the actual outcome result of a task is used to define the fluent in question.

Example 1 Consider the following successor state axiom for functional fluent $\mathit{At}(x, s)$ to keep track of the location of entities (e.g., actors and robots) in our emergency scenario:

$$\begin{aligned}&\mathit{At}(x, do(a, s)) = l \equiv\\ &\quad (\exists id, l_s, l_d)a = \text{FINISHED}(x, id, \text{GO}(l_s, l_d), l) \vee\\ &\quad [\mathit{At}(x, s) = l \wedge\\ &\quad (\neg\exists id, l')a = \text{FINISHED}(x, id, \text{GO}(l_s, l_d), l') \wedge l' \neq l].\end{aligned}$$

In words, service x is in location l if x was just released of a task GO whose actual physical outcome result was l. Observe that l, the new location of x, may happen to be different to the expected (destination) location l_d. ■

An important property of the domain in question is that one stating whether a service is connected to the network. Rather than defining a new data fluent, an abbreviation will be enough.

Example 2 The abbreviation $\mathit{Connected}(x, s)$ denotes that service x is within network connectivity range and is defined as follows:

$$\begin{aligned}&\mathit{Connected}(x, s) \stackrel{\text{def}}{=}\\ &\quad \mathit{Robot}(x) \vee \mathit{Covered}(\mathit{At}(x, s)) \vee\\ &\qquad \exists r.\mathit{Robot}(r) \wedge \mathit{Neigh}(\mathit{At}(x, s), \mathit{At}(r, s)).\end{aligned}$$

That is, service x is connected to the network iff x is actually robot (robots have their own connectivity), or x's location is covered by the main base network, or x is adjacent to a robot providing network connectivity in its surroundings. ■

Now, in KiPs, data fluents and abbreviations will be often used for defining the *preconditions* of domain tasks. By doing so, the PMS can reason, at run-time, about the active process instance relative to the current context.

Example 3 The following precondition axiom defines when the PMS can assign a navigation task to a service:

$$\begin{aligned}&\mathit{Poss}(\text{ASSIGN}(srv, id, \text{GO}(l_s, l_d), l_e), s) \equiv\\ &\quad (\mathit{At}(srv, s) = l_s) \wedge \mathit{Connected}(srv, s) \wedge\\ &\quad (l_e = l_d) \wedge \mathit{Capable}(srv, \text{GO}(l_s, l_d)).\end{aligned}$$

That is, the service (e.g., human actor, robot, etc.) ought to be at the source location l_s, connected to the network, and capable of locomotion. Moreover, the expected outcome of the task needs to be the destination location l_d. ■

This concludes the exposition of the first three aspects of a SmartPM action theory, as listed above.

Exception Monitoring We now turn our attention to the mechanism for automatically detecting failures/exceptions. To that end, we leverage on De Giacomo, Reiter, and Soutchanski (1998)'s technique of adaptation from the field of agent-oriented programming, by specializing it to our KiP setting. We consider adaptation as *reducing the gap* between the *expected reality*, the (idealized) model of reality, and the *physical reality*, the real world with the actual conditions and outcomes. A misalignment of the two realities stems from errors or exceptions in the tasks' outcomes, and may require explicit intervention.

The physical reality is captured using data fluents (together with their corresponding successor state axioms) as described above (e.g., fluent $\mathit{At}(x, s)$). The expected reality, in turn, is captured by a set of automatically generated fluents from the data fluents. Technically, for every fluent $F(\vec{x}, s)$, a new fluent $F_{\text{exp}}(\vec{x}, s)$ (F-expected) is used. Intuitively, $F_{\text{exp}}(\vec{x}, s)$ represents the value of $F(\vec{x}, s)$ in the "expected" (or "desired") execution. If $F(\vec{x}, do(a, s)) \equiv \Psi_F(\vec{x}, a, s)$ is F's successor state axiom, its expected version $F_{\text{exp}}(\vec{x}, s)$ is defined as follows:

$$\begin{aligned}&F_{\text{exp}}(\vec{x}, do(a, s)) \equiv\\ &\quad a = \text{ALIGN} \supset F(\vec{x}, s) \vee\\ &\quad [(\exists srv, id, t, \vec{o_r})a = \text{FINISHED}(srv, id, t, \vec{o_r}) \supset\\ &\qquad \Psi^*_F(\vec{x}, \text{FINISHED}(srv, id, t, \mathit{ExOut}(id, t, s)), s)] \vee\\ &\quad [a \neq \text{ALIGN} \wedge (\forall \vec{y})a \neq \text{FINISHED}(\vec{y}) \supset \Psi^*_F(\vec{x}, a, s)].\end{aligned}$$

where $\Psi^*_F(\vec{x}, a, s)$ is obtained by replacing each fluent X in $\Psi_F(\vec{x}, a, s)$ (the right-hand-side formula of F's successor state axioms) with its expected version X_{exp}.

The first disjunct states that the special action ALIGN assigns the actual value of the fluent to its expected value, thus providing a synchronization mechanism between the expected and physical realities. The remaining two disjuncts state together that the dynamics of F_{exp} is that of F *assuming all task outcomes turn out to be the ones expected*. Technically, this is achieved by replacing all data fluents in Ψ_F

with their expected version and the actual outcomes $\vec{o_r}$ of finished tasks with the expected ones (term $ExOut(id, t, s)$).

Example 4 The corresponding (syntactically simplified) expected version for fluent $At_{\text{exp}}(x, s)$ is as follows:

$$\begin{array}{l} At_{\text{exp}}(x, do(a, s)) = l \equiv \\ \quad a = \text{ALIGN} \supset (At(x, s) = l) \vee \\ \quad [(\exists id, l_s, l_d, l_r) a = \text{FINISHED}(x, id, \text{GO}(l_s, l_d), l_r) \supset \\ \qquad l = ExOut(id, \text{GO}(l_s, l_d), s)] \vee \\ \quad [At_{\text{exp}}(x, s) = l \wedge a \neq \text{ALIGN} \wedge \\ \qquad (\forall id, l_s, l_d, l_r) a \neq \text{FINISHED}(x, id, \text{GO}(l_s, l_d), l_r)]. \end{array}$$

Observe that if the action a denotes the completion of a GO task for x with some actual outcome l_r, the new location of x is that expected (denoted by $ExOut(id, \text{GO}(l_s, l_d), s)$). ■

Similarly, for each abbreviation $A(s) \stackrel{\text{def}}{=} \Psi(s)$, one can define corresponding abbreviation $A_{\text{exp}}(s) \stackrel{\text{def}}{=} \Psi^*(s)$, where $\Psi^*(s)$ is obtained by replacing each fluent (or abbreviation) X in $\Psi(s)$ with its expected version X_{exp} and all actual task outcomes $\vec{o_r}$ with their expected ones $ExOut(id, t, s)$. Of course, one can also define special expected fluent or abbreviations that can encode particular expectations. For example, by simply taking $Connected_{\text{exp}}(x, s) \stackrel{\text{def}}{=} \texttt{true}$, we are able to encode the constraint that actors are expected to always be connected.

So, using the data fluents and their expected versions, a misalignment can be recognized. However, it may only be important to check for mismatches among *some* properties of the world. So, we assume that the designer specifies distinguished abbreviation $Misaligned(s)$ to characterize misalignment situations requiring process adaptation. The general form of this abbreviation is as follows:

$$\begin{array}{l} Misaligned(s) \stackrel{\text{def}}{=} \\ \quad \exists \vec{x_1}.\Phi_{X^1}(\vec{x_1}, s) \supset \neg[X^1(\vec{x_1}, s) \equiv X^1_{\text{exp}}(\vec{x_1}, s)] \vee \\ \qquad \vdots \\ \quad \exists \vec{x_n}.\Phi_{X^n}(\vec{x_n}, s) \supset \neg[X^n(\vec{x_n}, s) \equiv X^n_{\text{exp}}(\vec{x_n}, s)], \end{array}$$

where $X^i(\vec{x_i}, s)$, with $i \in \{1, \ldots, n\}$, are all the data fluents and abbreviations used in the SmartPM application, and $\Phi_{X^i}(\vec{x_i}, s)$ states when fluent/abbreviation $X^i(\vec{x_i}, s)$ needs to be monitored for misalignment between its physical and expected values. We use $\Phi_{X^i}(\vec{x_i}, s) = \texttt{true}$ and $\Phi_{X^i}(\vec{x_i}, s) = \texttt{false}$ to specify permanent and no monitoring, respectively, for the corresponding fluent/abbreviation.

Example 5 Assuming we are only interested in monitoring the location of actors and their connectivity, we would specify the following definition:

$$\begin{array}{l} Misaligned(s) \stackrel{\text{def}}{=} \\ \quad \exists x_1.Actor(x_1) \supset \neg[At(x_1, s) \equiv At_{\text{exp}}(x_1, s)] \vee \\ \quad \exists x_1.Actor(x_1) \supset \\ \qquad \neg[Connected(\vec{x_1}, s) \equiv Connected_{\text{exp}}(\vec{x_1}, s)]. \end{array}$$

Observe the definition is not concerned about exceptions on the location of robots or their connectivity, for example. ■

This concludes the explanation on what type of situation calculus BAT we shall use in a SmartPM application. Let us call this theory $\mathcal{D}_{\text{SmartPM}}$.

SmartPM High-Level Program

Algorithm 1 shows the IndiGolog program for the SmartPM system. The program, as any high-level program, is meant to be executed relative to the basic action theory $\mathcal{D}_{\text{SmartPM}}$ as developed above, which shall give meaning to conditions and primitive statements in the program (i.e., actions).

The top-level part of PMS involves four interrupts running at different priorities, as long as the domain process is yet not finished. The highest two priority programs deal with automated process adaptation. First, if the system has just been adapted, then the two realities—expected and physical—must be aligned, as a new repair plan has been found and a new synchronization point has been reached. Second, the system checks for a misalignment between the two realities, as explained above. If a mismatch is recognized, the adaptation procedure is triggered (see below).

At medium priority, the PMS runs the IndiGolog program reflecting the actual KiP, represented by procedure **Process**() and corresponds to the process depicted in Figure 1(a). Finally, at the lowest priority (when the process cannot advance a step further) the PMS just waits for an exogenous action to arrive from one of the services (e.g., FINISHED signaling the completion of a task). While waiting, the (human) process designer could also manually intervene (for example, by adding new services or updating the capabilities of existing services).

Managing the life-cycle of a task instance—procedure **ManageExecution**()—involves selecting a free service capable of carrying it out, assigning the task to the chosen service, allowing the start of the service, acknowledging its completion and fully releasing the service from the task. Notice the use of the non-deterministic choice of argument $\pi srv.\delta(srv)$ to select an appropriate service.

The most interesting part of the procedure involves procedure **Adapt**(). An adaptation is very simple: *find a sequence of actions that will resolve the misalignment*. This is exactly what the code inside the search operator Σ does: *pick n actions (and execute them) zero, one, or more times such that abbreviation Misaligned*(s) *becomes false.* The action SETMUSTALIGN at the front of the search construct will just make $MustAlign(s)$ true, which will trigger (provided an actual adaptation plan is found) the top-priority program in the main procedure to force an alignment of the two realities. Observe that because the adaptation mechanism runs at higher priority than the actual process, the recovery plan found will be run *before* whatever part of the domain process remains to be executed.

While this specification of the automated adaptation procedure turns out to be extremely clean and simple, the direct use of the native search operator provided by the IndiGolog architecture (De Giacomo et al. 2009) poses serious problems in terms of efficiency. Indeed, the search operator provided by IndiGolog performs basic blind search and as a result is not able to cope with extremely easy adaptation tasks. The important observation to make is that while the search operator is meant to handle *any* IndiGolog high-level program (including ones containing nested search operators), our SmartPM system uses a specific program that encodes a (classical) planning problem. So, leveraging on

Algorithm 1: IndiGolog high-level program for PMS.

Proc **SmartPM**()
  $\langle \neg \mathit{Finished} \wedge \mathit{MustAlign} \rightarrow \text{ALIGN} \rangle \rangle$
  $\langle \neg \mathit{Finished} \wedge \mathit{Misaligned} \rightarrow \mathbf{Adapt}() \rangle \rangle$
  $\langle \neg \mathit{Finished} \rightarrow [\mathbf{Process}(); \text{FINISH}] \rangle \rangle$
  $\langle \neg \mathit{Finished} \rightarrow [\text{WAIT}] \rangle$.
Proc **Adapt**()
  $\Sigma_{\text{LPG}}[\text{SETMUSTALIGN}; (\pi a.a)^*; \neg \mathit{Misaligned}?]$.
Proc **ManageExecution**($\mathit{Task}, \mathit{id}, \mathit{ExpOut}$)
  $(\pi\ \mathit{srv})$.
    $(\mathit{Capable}(\mathit{srv}, \mathit{Task}) \wedge \mathit{Free}(\mathit{srv}))?$;
    ASSIGN($\mathit{srv}, \mathit{id}, \mathit{Task}, \mathit{ExpOut}$);
    START($\mathit{srv}, \mathit{id}, \mathit{Task}$);
    ACKCOMPL($\mathit{srv}, \mathit{id}, \mathit{Task}$);
    RELEASE($\mathit{srv}, \mathit{id}, \mathit{Task}$)

Proc **Process**()
  [**Branch1**()|| **Branch2**()|| **Branch3**()].
Proc **Branch1**()
  ManageExecution(GO($loc00, loc33$), $id_1, loc33$);
  ManageExecution(TAKEPHOTO($loc33$), id_2, ok);
  ManageExecution(UPDATESTATUS($loc33$), id_3, ok)
Proc **Branch2**()
  ManageExecution(GO($loc00, loc32$), $id_4, loc32$);
  ManageExecution(EVACUATE($loc32$), id_5, ok);
  ManageExecution(UPDATESTATUS($loc32$), id_6, ok)
Proc **Branch3**()
  ManageExecution(GO($loc00, loc31$), $id_7, loc31$);
  ManageExecution(EVACUATE($loc31$), id_8, ok);
  ManageExecution(UPDATESTATUS($loc31$), id_9, ok)

the recent progress of classical planning systems, we implement the search operator call in procedure **Adapt**(), by using an off-the-shelf planner to synthesise the recovery plan, and then fit such a plan back into the IndiGolog framework. In this work, we used the LPG-td system (Gerevini, Saetti, and Serina 2003), one of the many state-of-the-art planning systems available. The basic search scheme of LPG-td is inspired by Walksat (Selman, Kautz, and Cohen 1994), an efficient procedure for solving SAT-problems; as expected, it outperforms the blind search operator by several orders of magnitudes. Nonetheless, our approach is orthogonal to other planning systems.

We do not go over all the details on how the interface between IndiGolog and LPG-td has been implemented but just go over the main ingredients. First of all, given a BAT for a KiP application, the corresponding PDDL planning domain is built offline and stored. Because we are not concerned with the exogenous actions generated by services to acknowledge the start and termination of assigned processes (i.e., actions READYTOSTART and FINISHED), we do not model the full PMS assign-start-acknowledge-release task life-cycle, but just encapsulates them all in the actual name of the task being handled (e.g., GO). By doing that, we assume that the life-cycle of a task instance will follow its normal evolution. The SmartPM BAT will define a form of tasks and services repository, which may include entities not used in the current running process. So, the PDDL domain for our case study will contain, among others, the following action schema:

```
(:action go
 :parameters (?x - srv ?from - loc
              ?to - loc)
 :precondition (and (free ?x)
                (provides ?x movement)
                (at ?x ?from) (connected ?x))
 :effect (and
          (not (at ?x ?from)) (at ?x ?to)))
```

Note that the task-action in the planning system will contain the service in charge and its expected effects.

Then, whenever a search operator adaptation program is called in procedure **Adapt**(), the current situation of the IndiGolog system is encoded as a planning initial state (as the set of fluents that are true) and formula $\mathit{Misaligned}(s)$ is encoded as a goal state (by taking the collection of relevant fluents to be as their expected versions). Those two states, together with the planning domain already pre-computed, are then passed to the LPG-td system.

Finally, if the planner finds a plan that brings about the (desired) expected reality, such a plan—built from task names only—is translated into the typical assign-start-acknowledge-release task life-cycle IndiGolog program. As stated above, the plan will run *before* the actual domain process, which shall resume then, hopefully from the expected reality. In our running example, the full IndiGolog program would encode the KiP depicted in Figure 2(b).

This concludes the complete overview of SmartPM, built from a BAT and an IndiGolog high-level program that uses a classical planner for its specific (classical planning) search operator implementation. Let us now look at some empirical evaluation of such a system.

5 Validation

In order to investigate the feasibility of the SmartPM approach, we performed experiments to study the time required for synthesizing a recovery plan for different adaptation problems. We made our tests by using the LPG-td planner (Gerevini, Saetti, and Serina 2003). We chose LPG-td as *(i)* it treats most of PDDL 2.2 features; and *(ii)* it has been developed in two versions: a version tailored to computation speed, named LPG-td.speed, which produces *sub-optimal plans* that do not prove any guarantee other than the correctness of the solution, and a version tailored for *plan quality*, named LPG-td.quality. LPG-td.quality differs from LPG-td.speed basically in that it does not stop when the first plan is found, but continues until a stopping quality criterion is met. In our experiments, the criterion for plan quality was set to minimal plan length.

The experimental setup was performed with the test case shown in our train emergency running example. We used a tasks repository containing 20 different emergency management tasks, annotated with 28 relational predicates, 2 derived predicates, and 4 functional numeric fluents. We provided 185 different planning problems of different complexity, by manipulating ad-hoc the values of the initial state and the goal in order to devise adaptation problems of growing complexity. The results are reported in Table 1. Each row summarizes the results of various planning instances ($\#I$ instances). Column L-RP indicates the length of the shortest

Table 1: Performances of LPG-td. L-RP = length of the shortest recovery plan; #I = number of problem instances; AT-SOS/AL-SOS = average time/length for sub-optimal plan; and AT-QS = average time for quality plan.

L-RP	#I	AT-SOS	AL-SOS	AT-QS
1	29	6.769	3	7.768
2	36	7.213	3	16.865
3	32	7.846	4	24.123
4	25	8.128	5	37.017
5	21	8.598	8	39.484
6	17	8.736	9	52.421
7	13	9.188	13	73.526
8	12	9.953	14	81.414

plans for all instances. Column AL-SOS indicates the average number of actions of the sub-optimal solutions found by LPG-td.speed; whereas column AT-SOS reports the time it took to find such plan solutions. Those plans will generally include more tasks than the ones strictly needed. Finally, column AT-QS indicates the average time taken by LPG-td.quality to produce a quality plan. For example, on 21 different planning problems requiring a recovery procedure of length 5, the LPG-td planner is able to find, on average, a sub-optimal plan in 8.598 seconds (with an average of three tasks more) and a quality plan (of exactly 5 tasks necessary for recovery) in 39.484 seconds. Consequently, the approach is feasible for medium-sized dynamic processes used in practice.[4]

Effectiveness of SmartPM in Adapting Processes

An important aspect to consider during the development of a PMS with adaptation features concerns its *effectiveness* in supporting process models having control flows with different structures. We define effectiveness as the *ability of a PMS to complete the execution of a process model (i.e., to execute all the tasks involved in a path from the start event to the end event) by adapting automatically its running process instance if some failure/exception arises, without the need of any manual intervention of the process designer at run-time*.

To evaluate the effectiveness of SmartPM, we developed a software module named the SmartPM *Simulator*, which is able to automatically build IndiGolog processes and corresponding $\mathcal{D}_{\text{SmartPM}}$ theories, by simulating their execution on the basis of some customizable parameters:

- *Structure of the control flow*, with tasks organized in sequence or in 3 or 5 parallel branches.
- *Tasks repository size*, equal to 25, 50, or 75 tasks.
- *Number of available services* in the initial situation for task assignment. We fixed this value to 5 services.
- *Number of task preconditions/effects*; we allowed the generation of tasks having a maximum of 5 conditions in the precondition/effect axioms.
- *Number of available fluents*; we allowed the generation of 50 relevant data fluents that may assume boolean values. For each data fluent, the SmartPM Simulator automatically builds the corresponding expected fluent for monitoring possible tasks failures.
- *Percentage of tasks failures* This parameter may assume two possible values (30% or 70%), and affects the percentage of tasks error during the process execution.
- *Percentage of capabilities coverage* (30% or 70%), that affects the ability of each available service to execute the tasks stored in the repository.

Given *(i)* a specific structure of the control flow, *(ii)* a fixed percentage of capabilities coverage and *(iii)* of tasks failures, for each possible size of the tasks repository the SmartPM *Simulator* generated 100 process models with control flows composed respectively by 5 to 25 tasks randomly picked from the tasks repository. In total, we tested 3600 process models. Test results are shown in Figure 3. Collected data are organized in 4 diagrams obtained by combining the percentage of failures (FP) with the the percentage of capabilities coverage (SC). Each bar corresponds to the enactment of 100 different process models with a fixed size of the tasks repository and a specific structure of the control flow.

The analysis of the performed tests points out some interesting aspect. For example, let us consider the diagram in Figure 3(a), that shows the effectiveness of SmartPM in executing processes with a FP equal to 30% and a SC fixed to 30%. When the size of the tasks repository is equal to 75, the effectiveness of SmartPM in executing 100 process models composed by a sequence of tasks is equal to 82%. This means that 82 processes out of 100 that were executed have been correctly enacted and completed, while for the other 18 processes the system did not find any recovery plan for dealing with an exception occurrence. We can note that the effectiveness decreases if the instances have tasks organized in three parallel branches (79%) and in five parallel branches (75%). In general, the effectiveness of the SmartPM system decreases as the number of parallel branches increases, since more services are possibly involved at the same time for tasks execution, by letting only few services available for process adaptation and recovery. Furthermore: *(i)* when the FP increases, the effectiveness of the SmartPM system decreases; *(ii)* when the SC increases, the effectiveness of the SmartPM system increases, because there are more possibilities for a task in the repository to be selected by an available service.

To sum up, the execution of 3600 process models with different structures was a valid test for measuring the effectiveness of SmartPM, that was able to complete 2537 process instances, corresponding to an effectiveness of about 70.5%.

6 Related Work

The AI community has been involved with research on process management for several decades, and AI technologies can play an important role in the construction of PMS engines that manage complex processes, while remaining robust, reactive, and adaptive in the face of both environmental and tasking changes (Myers and Berry 1998). One of the

[4] We did our tests by using an Intel U7300 CPU 1.30GHz Dual Core, 4GB RAM machine.

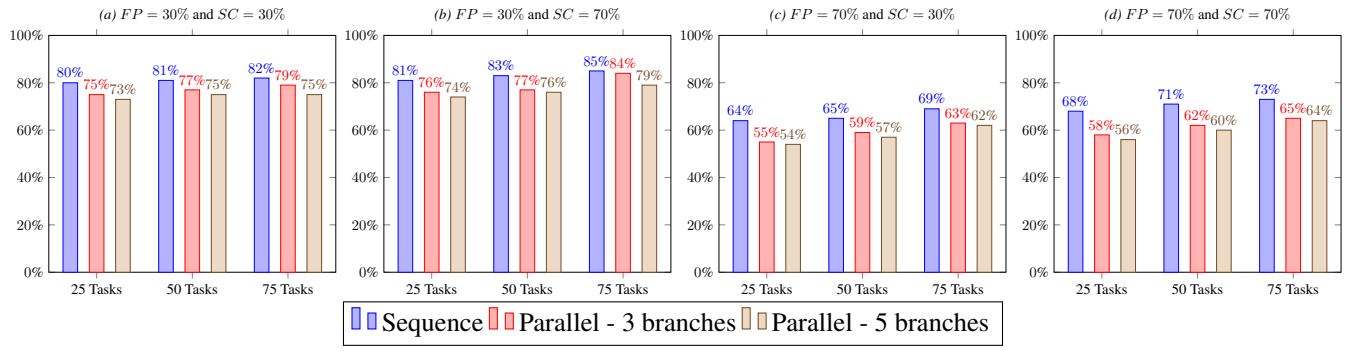


Figure 3: Analysis of the SmartPM effectiveness. The x-axis states the size of the tasks repository, while the y-axis indicates the effectiveness in executing process models with a specific structure of the control flow.

first works dealing with this research challenge is (Beckstein and Klausner 1999). It discusses at high level how the use of an intelligent assistant based on planning techniques may suggest compensation procedures or the re-execution of activities if some expected failure arises during the process execution. In (Jarvis et al. 1999) the authors describe how planning can be interleaved with process execution and plan refinement, and investigates plan patching and plan repair as means to enhance flexibility and responsiveness.

A goal-based approach for enabling automated process instance change in case of emerging exceptions is shown in (Gajewski et al. 2005). If a task failure occurs at run-time and leads to a process goal violation, a multi-step procedure is activated. It includes the termination of the failed task, the sound suspension of the process, the automatic generation (through the use of a partial-order planner) of a new complete process definition that complies with the process goal and the adequate process resumption. A similar approach is proposed in (Ferreira and Ferreira 2006). The approach is based on learning business activities as planning operators and feeding them to a planner that generates a candidate process model that is able of achieving some business goals. If an activity fails during the process execution at run-time, an alternative candidate plan is provided on the same business goals. The major issue of (Gajewski et al. 2005; Ferreira and Ferreira 2006) lies in the replanning stage used for adapting a faulty process instance. In fact, it forces to completely re-define the process specification at run-time when the process goal changes (due to some activity failure), by completely revolutionizing the work-list of tasks assigned to the process participants (that are often humans). On the contrary, our approach adapts a running process instance by modifying only those parts of the process that need to be changed/adapted and keeps other parts stable.

A work dealing with process interference is that of van Beest et al. (2014). Process interference is a situation that happens when several concurrent business processes depending on some common data are executed in a highly distributed environment. During the processes execution, it may happen that some of these data are modified causing unexpected or wrong business outcomes. To overcome this limitation, the work van Beest et al. proposes a run-time mechanism which uses *(i) Dependency Scopes* for identifying critical parts of the processes whose correct execution depends on some shared variables; and *(ii) Intervention Processes* for solving the potential inconsistencies generated from the interference, which are automatically synthesised through a domain independent planner based on constraint techniques. While closely related to van Beest's work, our account deals with changes in a more abstract and domain-independent way, by just checking misalignment between expected/physical realities. Conversely, van Beest's work requires specification of a (domain-dependent) adaptation policy, based on volatile variables and when changes to them become relevant.

7 Conclusion

As discussed in this paper, SmartPM defines a general approach, a concrete framework and a prototype PMS for automated adaptation of processes. Our purpose was to demonstrate that the combination of procedural and imperative models with declarative elements, along with the exploitation of techniques from the field of AI such as situation calculus, IndiGolog and classical planning, can increase the ability of existing PMSs of supporting and adapting KiPs.

The adaptation mechanism is based on execution monitoring for detecting failures and context changes at run-time, without requiring to predefine any specific adaptation policy or exception handler at design-time (as most of the current approaches do). The use of classical planning techniques for the synthesis of the recovery procedure has a twofold consequence. On the one hand, we can exploit the good performance of current state-of-the-art planners to solve medium-sized real-world problems as used in practice. On the other hand, classical planning imposes some restrictions for addressing more expressive problems, including incomplete information, preferences and multiple task effects.

We note that, besides task failures, the execution of a KiP could be also jeopardized by the occurrence of *exogenous events*—e.g., a sudden fire outbreak in a coach—that could change, in asynchronous manner, some *contextual* properties of the scenario in which the process is under execution, hence possibly requiring the KiP to be adapted. Future work

is required to extend our approach to deal such disruptive exogenous events.

The current prototype of SmartPM is developed to be effectively used by process designers and practitioners.[5] Users define processes in the well-known BPMN language, enriched with semantic annotations for expressing properties of tasks, which allow our interpreter to derive the IndiGolog program representing the process. Interfaces with human actors (as specific graphical user applications in Java) and software services (through Web service technologies) allow the core system to be effectively used for enacting processes. Our future work will also improve the current prototype in order to be compliant with many other technologies adopted by process practitioners, e.g., RESTful services and HTML-based user interfaces.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. The first two authors were supported by Sapienza - Universitá di Roma through the grant SUPER and by Regione Lazio and FILAS through the "Progetto Sensoristica Interconnessa per la Sicurezza". The last author was supported by the Australian Research Council under a Discovery Project DP120100332.

References

Beckstein, C., and Klausner, J. 1999. A Meta Level Architecture for Workflow Management. *Journal of Integrated Design and Process Science* 3(1):15–26.

De Giacomo, G.; Lespérance, Y.; Levesque, H.; and Sardina, S. 2009. IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents. In *Multi-Agent Programming*. Springer US. 31–72.

De Giacomo, G.; Reiter, R.; and Soutchanski, M. 1998. Execution Monitoring of High-Level Robot Programs. In *6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, 453–465.

Di Ciccio, C.; Marrella, A.; and Russo, A. 2012. Knowledge-intensive Processes: An Overview of Contemporary Approaches. In *1st International Workshop on Knowledge-intensive Business Processes (KiBP'12)*, 33–47.

Dumas, M.; van der Aalst, W. M. P.; and ter Hofstede, A. H. M. 2005. *Process-aware Information Systems: Bridging People and Software Through Process Technology*. Wiley Online Library.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Technical report, Albert-Ludwigs-Universität Freiburg, Institut für Informatik.

Ferreira, H., and Ferreira, D. 2006. An Integrated Life Cycle for Workflow Management Based on Learning and Planning. *International Journal of Cooperative Information Systems* 15(4):485–505.

Gajewski, M.; Meyer, H.; Momotko, M.; Schuschel, H.; and Weske, M. 2005. Dynamic Failure Recovery of Generated Workflows. In *16th International Conference on Database and Expert Systems Applications (DEXA'05)*, 982–986.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning Through Stochastic Local Search and Temporal Action Graphs in LPG. *Journal of Artificial Intelligence Research* 20:239–290.

Helal, S.; Mann, W.; El-Zabadani, H.; King, J.; Kaddoura, Y.; and Jansen, E. 2005. The Gator-Tech Smart House: A Programmable Pervasive Space. *Computer* 38(3):50–60.

Jarvis, P.; Moore, J.; Stader, J.; Macintosh, A.; du Mont, A. C.; and Chung, P. 1999. Exploiting AI Technologies to Realise Adaptive Workflow Systems. In *AAAI Workshop on Agent-Based Systems in the Business Context*, 25–34.

Lenz, R., and Reichert, M. 2007. IT support for healthcare processes - premises, challenges, perspectives. *Data Knowledge Engineering* 61(1):39–58.

Marrella, A.; Russo, A.; and Mecella, M. 2012. Planlets: Automatically Recovering Dynamic Processes in YAWL. In *20th International Conference on Cooperative Information Systems (CoopIS'12)*, 268–286.

Myers, K., and Berry, P. 1998. Workflow Management Systems: An AI Perspective. *AIC-SRI report*.

Nau, D.; Ghallab, M.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Reichert, M., and Weber, B. 2012. *Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies*. Springer.

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

Sadiq, S. W.; Sadiq, W.; and Orlowska, M. E. 2001. Pockets of Flexibility in Workflow Specification. In *20th International Conference on Conceptual Modeling: Conceptual Modeling (ER'01)*, 513–526.

Selman, B.; Kautz, H. A.; and Cohen, B. 1994. Noise Strategies for Improving Local Search. In *12th National Conference on Artificial Intelligence (AAAI-94)*, 337–343.

van Beest, N.; Kaldeli, E.; Bulanov, P.; Wortmann, J.; and Lazovik, A. 2014. Automated Runtime Repair of Business Processes. *Information Systems* 39:45–79.

van der Aalst, W. M. P.; ter Hofstede, A. H. M.; Kiepuszewski, B.; and Barros, A. P. 2003. Workflow patterns. *Distributed Parallel Databases* 14(1):5–51.

van der Aalst, W. M. P.; ter Hofstede, A. H. M.; and Weske, M. 2003. Business Process Management: A Survey. In *1st International Conference on Business Process Management (BPM'03)*, 1–12.

Weske, M. 2012. *Business Process Management: Concepts, Languages, Architectures*. Springer.

[5] See `www.dis.uniroma1.it/~smartpm`