

Intelligent Process Adaptation in the SmartPM System

ANDREA MARRELLA and MASSIMO MECELLA, Sapienza Università di Roma, Italy
SEBASTIAN SARDINA, RMIT University, Australia

The increasing application of process-oriented approaches in new challenging dynamic domains beyond business computing (e.g., healthcare, emergency management, factories of the future, home automation, etc.) has led to reconsider the level of flexibility and support required to manage complex knowledge-intensive processes in such domains. A knowledge-intensive process is influenced by user decision making and coupled with contextual data and knowledge production, and involves performing complex tasks in the “physical” real world to achieve a common goal. The physical world, however, is not entirely predictable, and knowledge-intensive processes must be robust to unexpected conditions and adaptable to unanticipated exceptions, recognizing that in real-world environments it is not adequate to assume that all possible recovery activities can be predefined for dealing with the exceptions that can ensue. To tackle this issue, 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 when unanticipated exceptions occur, 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. The approach, which is backed by a formal model, has been implemented and validated with a case study based on real knowledge-intensive processes coming from an emergency management domain.

Categories and Subject Descriptors: Applied Computing [**Enterprise computing**]: Business process management—*Business process management systems*; Computing methodologies [**Artificial intelligence**]: Knowledge representation and reasoning; Computing methodologies [**Artificial intelligence**]: Planning and scheduling

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Classical planning, IndiGolog, Knowledge-intensive processes, Pervasive applications, Process adaptation, Process modeling and execution, Situation calculus

ACM Reference Format:

Andrea Marrella, Massimo Mecella and Sebastian Sardina. 2016. Intelligent Process Adaptation in the SmartPM System *ACM Trans. Intell. Syst. Technol.* V, N, Article 0 (2016), 57 pages.
DOI: <http://dx.doi.org/10.1145/0000000.0000000>

1. INTRODUCTION

This paper is concerned with the use of three well-established *Knowledge Representation and Reasoning* (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 2013], an active area of research that is highly relevant from a practical point of view while offering many technical challenges for computer scientists and researchers.

BPM solutions have been prevalent in both industry products and academic prototypes since the late 1990s [Weske 2012]. 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 et al. 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. The representation of a single execution of a process model is called a *process instance* [Dumas et al. 2013].

Current maturity of process management methodologies has led to the application of process-oriented approaches in new rich challenging domains beyond business computing, such as healthcare [Lenz and Reichert 2007], emergency management [Marrella et al. 2012], factories of the future [Seiger et al. 2014] and home automation [Helal et al. 2005]. In addition to this, the current generation of mobile devices and their wireless communication capabilities have become useful to support mobile workers, allowing them to execute processes and tasks in dynamic environments. 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 et al. 2014]—genuinely knowledge and data centric—and require the integration of the data dimension with the traditional control flow dimension by considering them as a whole.

During the enactment of KiPs, variations or divergence from structured reference models are common due to exceptional circumstances arising (e.g., autonomous user decisions, exogenous events, or contextual changes), thus requiring the ability to properly *adapt* the process behavior. According to [Sadiq et al. 2001], *Process Adaptation* can be seen as the ability of a process to react to exceptional circumstances (that may or may not be foreseen) and to adapt/modify its structure accordingly. *Exceptions* are generally defined as distinct identifiable events which occur at a specific point in time during the execution of a process instance and result in deviations from normal execution arising during a business process [Russell et al. 2006].

Exceptions can be either *anticipated* or *unanticipated*. An anticipated exception can be planned at design-time and incorporated into the process model, i.e., a (human) process designer can provide an *exception handler* that is invoked during run-time to cope with the exception. Conversely, *unanticipated exceptions* generally refer to situations, unplanned at design-time, that may emerge at run-time and can be detected only during the execution of a process instance, when a mismatch between the computerized version of the process and the corresponding real-world process occurs. To cope with those exceptions, a PMS is required to allow *ad-hoc process changes* for adapting running process instances in a situation- and context-dependent way.

In knowledge-intensive scenarios, the fact is that the number of possible anticipated exceptions is often too large, and traditional manual implementation of exception handlers at design-time is not feasible for the process designer, that has to anticipate all potential problems and ways to overcome them in advance [Reichert and Weber 2012]. Furthermore, anticipated exceptions cover only partially relevant situations, as in knowledge-intensive scenarios many unanticipated exceptional circumstances may arise during the process instance execution. Therefore, the process designer often lacks the needed knowledge to model all the possible exceptions at the outset, or this knowledge can become obsolete as process instances are executed and evolve, by making useless her/his initial effort.

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. The general idea builds on the dualism between an *expected reality* and a *physical reality*: process execution steps and exogenous events have an impact on the physical reality and any deviation from the expected reality results in a mismatch (or exception) to be removed to allow process progression. 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 et al. 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’ 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 to adapt the faulty process instance. We refer to this adaptive framework for KiPs as SmartPM (Smart Process Management), described in Section 4.

The SmartPM framework has been implemented by relying on an existing IndiGolog interpreter [De Giacomo et al. 2009] and the state-of-the-art planning system LPG-td [Gerevini et al. 2004]. Furthermore, a graphical tool that allows non-experts in AI entering knowledge on processes has been provided. The sum of these components is the SmartPM implemented system, whose architecture is described in Section 5.

Besides providing the conceptual framework and the system architecture, we validated the approach with a case study based on real KiPs coming from an emergency management domain (Section 6). In Section 7, we position SmartPM with respect to the existing state-of-the-art adaptive PMSs and we compare it with other works that exploit AI techniques for enhanced process adaptation. Then, in Section 8 we conclude by providing a critical discussion about the general applicability of the SmartPM approach in dynamic domains and by tracing future work.

The choice of adopting AI technologies, and in particular those provided by the KR&R field, is motivated by their ability to provide the right abstraction level needed when dealing with dynamic situations in which data (values) play a relevant role in system enactment and automated reasoning over the system progress. In the field of BPM, many other formalisms and technologies are being used, such as Petri Nets [van Der Aalst 1996], Coloured Petri Nets [Jensen and Kristensen 2009], Workflow Nets [van der Aalst 1998], YAWL nets [ter Hofstede et al. 2009], BPMN [BPMI.org and OMG 2011] and process algebras [Puhmann and Weske 2005], with varying degrees of automated reasoning support over them. While Petri Nets and Workflow Nets do not support data-based decisions as well as data-driven execution of any kind due to the lack of data-awareness [Meyer et al. 2011], other formalisms such as Coloured Petri Nets, YAWL Nets, BPMN and Process Algebras are potentially all fine solutions for realizing our framework. However, the level of abstraction provided for manipulating data values and reasoning over dynamic changes is not formally specified (in the case of YAWL), performed at shallow level (in the case of BPMN) or at very low level (in the case of Coloured Petri Nets and Process Algebras), since such formalisms mainly focus on the control-flow perspective of a business process. Conversely, the KR&R field is rich of algorithms and systems that support the user in the creation, acquisition, adaptation, evolution, and sharing of data knowledge for specifying and implementing dynamic systems [Reichgelt 1991; Brachman and Levesque 2004; Reiter 2001]. As we will see in the further, the choice of KR&R technologies allows us to develop a very

clean and simple-to-manage framework for process adaptation based on relevant data manipulated by the process, without compromising efficiency and effectiveness of the proposed solution.

This paper extends our previous work presented in [Marrella et al. 2014] in various directions. First of all, we revise our original framework to deal with exogenous events as possible source of exception, a relevant aspect that was neglected in our previous work. This has required a substantial rework of the formal approach, to consider aspects such as the analysis of the impact of exogenous events on the contextual domain, which possibly entails the abortion of some running tasks for the maintenance of data consistency. Second, we insert a formal result showing that the “triggering” of adaptation performed by SmartPM is correct (cf. Theorem 4.9). Third, we describe the system’s architecture and implementation, providing details on the system components, their technological features and their relationships. Fourth, we provide a more realistic running example and more extended explanations of the whole approach. Fifth, the related work section has been extended significantly to cover the state of the art of adaptive PMSs. Sixth, we provide more experimental tests, performed with two different state-of-the-art planners, to synthesize recovery plans for different adaptation problems. Finally, we provide an appendix that shows the basic ingredients for modeling the process knowledge and the contextual properties of a dynamic environment in SmartPM, with special emphasis on the implemented user interface that assists the process designer in the definition of such a knowledge.

Before describing the technical proposal, we first provide an overview of our case study (Section 2) and some preliminary notions (Section 3) necessary to understand the rest of the paper.

2. CASE STUDY

Processes that are inherently knowledge-intensive can be found in several fields and domains. Human resource management, implementation projects, patient case management in hospitals, criminal investigations, domotics are all examples of domains that were subject to case studies and have been considered for the definition of scenarios and use cases for KiPs (for example, in [Lenz and Reichert 2007; Helal et al. 2005; Mundbrod et al. 2013]).

According to recent research studies investigating the role of KiPs in BPM [Kemsley 2011; Rosenfeld 2011; Harrison-Broninski 2013; Di Ciccio et al. 2014], almost all the classes of business processes may include elements that make them knowledge-intensive. The knowledge dimension may emerge, for example, in the way knowledge workers deal with unexpected exceptions in *well structured processes*, which reflect highly predictable routine work with low flexibility requirements and controlled interactions among process participants (such as administrative processes). Conversely, in *loosely structured or unstructured work practices*, the knowledge intensity lies in the way knowledge workers put in place their experience and expertise (that can not be easily formalized or shared through a well structured process) to the definition of the best course of actions. No explicit process model can be associated to unstructured processes, which are typically tied to the scope of groupware systems [Ellis et al. 1991] and are therefore outside the scope of our paper.

Conversely, we frame our discussion on a specific class of KiPs representing *structured processes with ad hoc exceptions* [Di Ciccio et al. 2014]. They have similar characteristics than structured processes, as they reflect operational activities that typically comply with a predefined plan. Although, the occurrence of external events and exceptions can make the structure of the process less rigid. The actual course of action may deviate from the predefined reference work practices and process adaptation strategies (that are not known in advance and may not be known until the time that the process

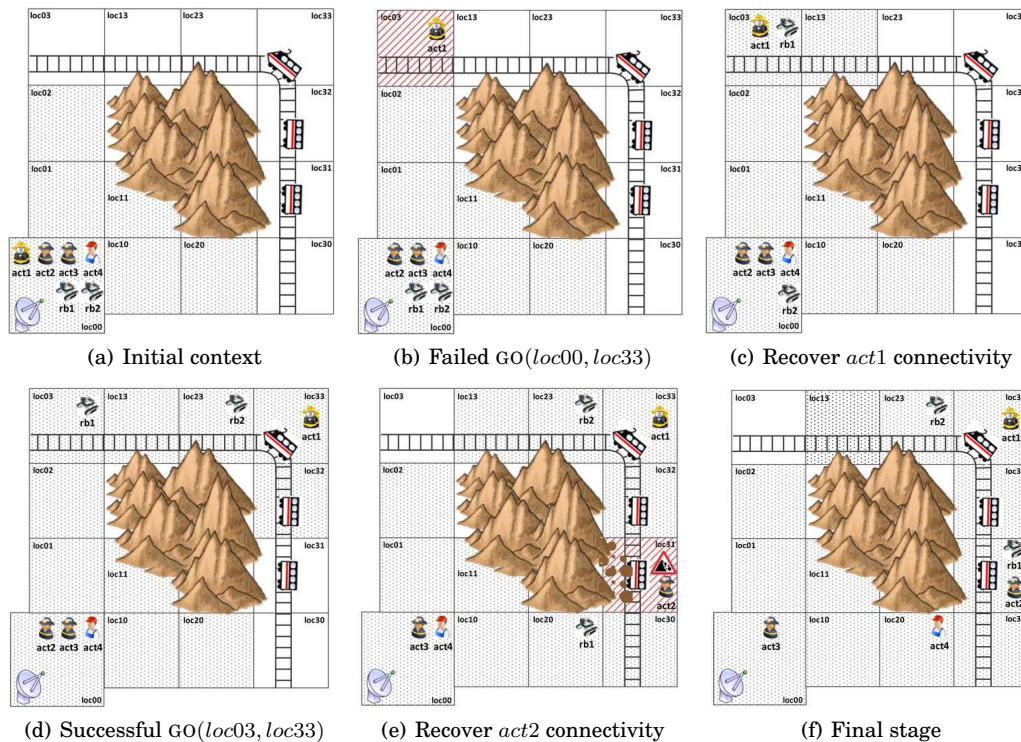


Fig. 1. A train derailment situation; area and context of the intervention.

has started execution) may be required. Several real-world processes belong to this class of KiPs; for example, processes for automotive, traffic management, smart manufacturing, emergency management, and consumer appliances [Lee 2008; Rajkumar et al. 2010]. The trend of managing KiPs in such domains has been fueled by the increased availability of sensors disseminated in the world that has led to the possibility to monitor in detail the evolution of the real-world objects of interest. The knowledge extracted from such objects allows to depict the contingencies and the context in which KiPs are carried out (including “technical” aspects like device capability constraints, wireless networking, device mobility, etc.), by consenting a fine-grained monitoring, mining, and decision support for them.

To make our discussion more concrete, our case study and evaluation involves a disaster management inspired by the European project WORKPAD.¹ [Capata et al. 2008; Humayoun et al. 2009a; 2009b; Marrella et al. 2011; Catarci et al. 2013] The proposed example attempts at highlighting, in an accessible manner, the key ingredients and ideas of the proposed approach in a simplified setting. Specifically, let us consider 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 $loc31$, respectively).

¹The WORKPAD project investigates how the use of a process-oriented approach can enhance the level of collaboration provided to first responders that act in emergency scenarios. The official web page of the project at: <http://www.dis.uniroma1.it/~workpad>

0:6

Andrea Marrella, Massimo Mecella, Sebastian Sardina

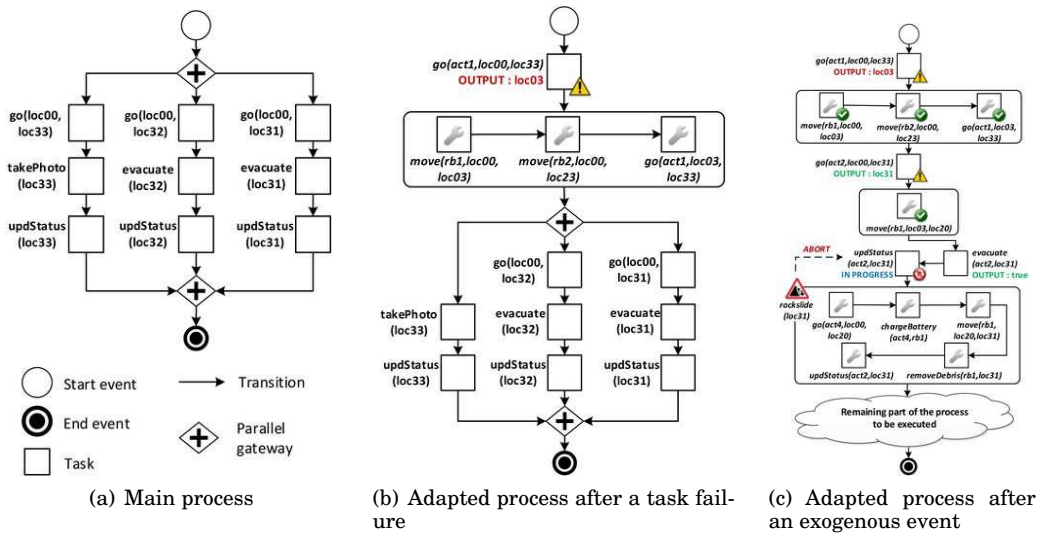


Fig. 2. An emergency response plan and its adaptation.

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 runtime 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 Such a process is composed of three parallel branches, with tasks instructing first responders to act for evacuating people from train coaches in *loc31* and *loc32*, taking pictures of the locomotive, and assessing the gravity of the accident (through the task UPDATESTATUS).

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

²See www.omg.org/spec/BPMN/

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 desired one, and most seriously, is out of the network connectivity range (cf. 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* (cf. 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, task $GO(loc03, loc33)$ is reassigned to actor *act1* (cf. 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. For example, in the third parallel branch, actor *act2* can now be instructed to reach *loc31*. However, even if *act2* completes its task as expected (cf. Figure 1(e)), a further exception is thrown. In fact, *act2* is out of the network connectivity range and, again, the PMS may instruct the first robot to move from cell *loc03* to cell *loc20* in order to re-establish network connection to actor *act2* (cf. top of Figure 1(c)). At this point, *act2* may start evacuating people from *loc31*.

We note that the execution of a KiP can also be jeopardized by the occurrence of *exogenous events*. Indeed, exogenous events 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 accordingly. Moreover, an exogenous event may possibly lead to the forced termination of a running task. For example, suppose that a rock slide collapses in location *loc31* (cf. Figure 1(e)) while *act2* is evaluating the damages in that area (i.e., *act2* is executing the $UPDATESTATUS(loc31)$ task). In such a case, the PMS needs first to abort the running task $UPDATESTATUS(loc31)$ (the presence of a rock slide may possibly prevent the correct execution of the task), and then to find a recovery procedure that allows to remove the rock slide from *loc31* by maintaining all the process participants inter-connected to the network. A possible solution is shown in Figure 1(f), and consists of instructing *act4* to reach *loc20* for recharging the battery of *rb1*, of moving the robot *rb1* in *loc31* in order to remove debris, and finally of reassigning the $UPDATESTATUS(loc31)$ task to *act2*. The corresponding adapted process is shown in the bottom of Figure 1(c).

The point is that it is not adequate to assume that the process designer can pre-define all possible recovery activities for dealing with unanticipated exceptions and exogenous events 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

The proposed approach is based on KR&R methods and technologies, namely situation calculus, IndiGolog and classical planning. Before providing a self-contained concise introduction to them, we would like to highlight the motivations for such choices instead of more traditional business process execution engines. Many other formalisms (in particular Petri Nets-based and process algebras) have been successfully adopted for process management, but all of them are somehow based on synthesis techniques of the control-flow, when considering their automated reasoning capabilities. This implies the level of abstraction over dealing with data and dynamic situations is quite

raw, when compared with KR&R methods in which automated reasoning over data values and situations is much more developed. Clearly all approaches, in the end, potentially can lead to an automated adaptation framework; our argument is that the use of KR&R methods allows us to reach such an ambitious goal through a very clean and simple-to-manage framework, as presented in the following, without compromising the efficiency of the solution (which in general may be an issue when adopting KR&R technologies).

Situation Calculus and Basic Action Theories. 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 . A situation s is a sub-situation of s' , denoted $s \sqsubseteq s'$, iff s is a prefix of s' . Formally, the relation is axiomatized as follows: $s \sqsubseteq s' \equiv [s = s' \vee (\exists a, s'').s' = do(a, s'') \wedge s \sqsubseteq s'']$.

Example 3.1. The action $OPEN(x)$ is used to open a door x . The situation term $do(OPEN(d_2), do(OPEN(d_1), S_0))$ denotes the situation resulting from first opening door d_1 in S_0 and then opening door d_2 . \square

In situation calculus, relations and functions whose value may change from one situation to the next are modeled by means of so-called *fluents*. Technically, relations whose truth values vary from situation to situation are called *relational fluents*. They are denoted by predicate symbols taking a situation term as their last argument. Similarly, functions whose values vary from situation to situation are called *functional fluents*, and are denoted by function symbols taking a situation term as their last argument. For example, the relational fluent $DoorOpen(x, s)$ may denote that door x is open in situation s , whereas the functional fluent $Floor(x, s)$ may denote the floor number that elevator x is at in situation s . A special predicate $Poss(a, s)$ is used to state that action a is executable in situation s , whereas special (situation-independent) predicate $Exog(a)$ is used to denote that a is an exogenous event (i.e., an event originating from the external environment). 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:

- Σ . domain-independent foundational axioms to describe the structure of situations and some auxiliary relations like \sqsubseteq and $Executable(s)$;
- \mathcal{D}_{ss} . one successor state axiom per fluent capturing the effects and non-effects (i.e., *frame*) of actions;
- \mathcal{D}_{poss} . one precondition axiom per action specifying when the action is executable;
- \mathcal{D}_{una} . unique name axioms for actions; and
- \mathcal{D}_{S_0} . initial state axioms describing what is true initially in S_0 , as well as auxiliary situation independent information, such as axioms for predicate $Exog(a)$.

In particular, the *successor state axiom* for a relational 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 character-

³Free variables are assumed to be universally quantified.

izing the dynamics of fluent $F(\vec{x}, s)$. When $F(\vec{x}, s)$ is a functional fluent, its successor state axiom has the form $[F(\vec{x}, do(a, s)) = v] \equiv \Gamma_F(\vec{x}, a, v, s)$, where $\Gamma_F(\vec{x}, a, v, s)$ states that the fluent takes value v when action a is executed in situation s and satisfies the functional constraint $\models (\forall \vec{x}, a, s) \exists v. \Gamma_F(\vec{x}, a, v, s) \wedge (\forall v'). v' \neq v \supset \neg \Gamma_F(\vec{x}, a, v', s)$. Importantly, $\Psi_F(\vec{x}, a, s)$ and $\Gamma_F(\vec{x}, a, v, s)$ can accommodate [Reiter 2001]'s solution to the frame problem.⁴

Example 3.2. The successor state axioms for relational fluent $DoorOpen(x, s)$ and functional fluent $Floor(x, s)$ are as follows:

$$\begin{aligned} DoorOpen(x, do(a, s)) &\equiv \\ &a = OPEN(x) \wedge \neg Locked(x, s) \wedge \neg DoorOpen(x, s) \vee \\ &[DoorOpen(x, s) \wedge a \neq CLOSE(x)]; \\ Floor(x, do(a, s)) = v &\equiv \\ &(a = UP(x) \wedge v = Floor(x, s) + 1) \vee (a = DOWN(x) \wedge v = Floor(x, s) - 1) \vee \\ &[Floor(x, s) = v \wedge a \neq UP(x) \wedge a \neq DOWN(x)]. \end{aligned}$$

That is, a door x is open in situation $do(a, s)$ iff a denotes the action of opening x (that is closed and not locked⁵ in s), or x is already open in s and a is not the action of closing it. Similarly, elevator x is in floor v if it was in floor $v - 1$ ($v + 1$) after moving up (down) one floor, or it is already in floor v and the action just executed is not one moving up or down such elevator. \square

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 . Using $Poss$, we can define what it means for a situation s to be executable, that is, every action is possible, using the following definition:

$$Executable(s) \equiv s = S_0 \vee (\exists a, s'). s = do(a, s') \wedge Poss(a, s') \wedge Executable(s').$$

Example 3.3. The action of opening a door in situation s is possible only if the door is closed in s , and an elevator can go down only if it is not in the first floor:

$$\begin{aligned} Poss(OPEN(x), s) &\equiv \neg DoorOpen(x, s); \\ Poss(DOWN(x), s) &\equiv (Floor(x, s) > 1). \end{aligned}$$

\square

Finally, \mathcal{D}_{S_0} is a collection of first-order sentences whose only situation term mentioned is the situation constant S_0 . For example, if sentence $\forall x. Floor(x, S_0) = 1$ is included in \mathcal{D}_{S_0} , then it represents the fact that all elevators are (parked) in the first floor *initially*.

The IndiGolog high-level language. 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

⁴In AI, the frame problem is the challenge of capturing the effects of actions in a succinct way, without having to represent explicitly a large number of intuitively obvious non-effects [Reiter 2001].

⁵The relational fluent $Locked(x, s)$ denotes that door x is locked in situation s .

0:10

Andrea Marrella, Massimo Mecella, Sebastian Sardina

$\Sigma(\delta)$ —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 a BAT, 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
$\delta_1 \parallel \delta_2$	concurrency
$\delta_1 \gg \delta_2$	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 $\delta_1 \parallel \delta_2$ and $\delta_1 \gg \delta_2$; 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). The interrupt construct $\langle \phi \rightarrow \delta \rangle$ states that program δ ought to be executed to completion if ϕ happens to become true. Let's focus on it:

$$\langle \phi \rightarrow \delta \rangle \stackrel{\text{def}}{=} \mathbf{while} \textit{Interrupts_running} \mathbf{do}$$

$$\mathbf{if} \phi \mathbf{then} \delta \mathbf{else} \textit{false?} \mathbf{endif}$$

$$\mathbf{endWhile}$$

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 from higher priority processes, suspending any lower priority processes that may have been advancing, it repeatedly executes δ until ϕ becomes false. Once the interrupt body δ completes its execution, the suspended lower priority processes may resume. The control release also occurs if ϕ cannot progress (e.g., since no action meets its precondition).

IndiGolog programs are meant to be executed *online*, one step at a time, with nondeterministic choices resolved arbitrarily. Because of that, there is no guarantee a programs will terminate successfully and dead-end blocking situations (e.g., an action not being possible) may be reached. To deal with that and the fact that backtracking actions executed in the real world may not always be an option, IndiGolog incorporates the so-called search construct $\Sigma(\delta)$, which performs lookahead reasoning on δ to guarantee that a full, terminating, execution of δ will be eventually achieved. In concrete, every step performed on program δ will be one that is part of a terminating execution (see below for its formal semantics).

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 reason-

ing is required. Both aspects will end up being fundamental for our adaptive process management framework in the next section.

Formally, the semantics of IndiGolog is specified in terms of single-step transitions, using the following two predicates ([De Giacomo et al. 2009]):

- $Trans(\delta, s, \delta', s')$, which holds if one step of program δ in situation s may lead to situation s' with δ' remaining to be executed; and
- $Final(\delta, s)$, which holds if program δ may legally terminate in situation s .

For example, the axioms for a primitive action a , sequence, and non-deterministic choice of programs, concurrency, and lookahead search are as follows:

Primitive action:

$$\begin{aligned} Trans(a, s, \delta', s') &\equiv s' = do(a, s) \wedge Poss(a, s) \wedge \delta' = nil \\ Final(a, s) &\equiv false \end{aligned}$$

Sequence:

$$\begin{aligned} Trans(\delta_1; \delta_2, s, \delta', s') &\equiv \\ Trans(\delta_1, s, \delta'_1, s') \wedge \delta' &= \delta'_1; \delta_2 \vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s') \\ Final(\delta_1; \delta_2, s, \delta', s') &\equiv Final(\delta_1, s) \wedge Final(\delta_2, s) \end{aligned}$$

Non-deterministic choice of programs:

$$\begin{aligned} Trans(\delta_1 \mid \delta_2, s, \delta', s') &\equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s') \\ Final(\delta_1 \mid \delta_2, s) &\equiv Final(\delta_1, s) \vee Final(\delta_2, s) \end{aligned}$$

Concurrency:

$$\begin{aligned} Trans(\delta_1 \parallel \delta_2, s, \delta', s') &\equiv \\ Trans(\delta_1, s, \delta'_1, s') \wedge \delta' &= \delta'_1 \parallel \delta_2 \vee Trans(\delta_2, s, \delta'_2, s') \wedge \delta' = \delta_1 \parallel \delta'_2 \\ Final(\delta_1 \parallel \delta_2, s) &\equiv Final(\delta_1, s) \wedge Final(\delta_2, s) \end{aligned}$$

Lookahead search:

$$\begin{aligned} Trans(\Sigma(\delta), s, \delta', s') &\equiv \\ Trans(\delta, s, \delta'', s') \wedge \delta' &= \Sigma(\delta'') \wedge (\exists \delta_f, s_f). Trans^*(\delta'', s'', \delta_f, s_f) \wedge Final(\delta_f, s_f) \\ Final(\Sigma(\delta), s) &\equiv Final(\delta, s) \end{aligned}$$

Here, $Trans^*$ stands for the reflexive transitive closure of $Trans$. Observe that a single transition on a program $\Sigma(\delta)$ is one that can be eventually extended, via a sequence of follow-up transitions, to a final configuration. Importantly, the lookahead construct is propagated to the next remaining program, which guarantees that every future single step will also have such property: every step of $\Sigma(\delta)$ is “safe” and leads to a successful run of program δ .

Using $Trans$ and $Final$, one can formalize the notion of *online execution* for a given initial program δ_0 from an initial situation S_0 as a sequence of so-called configurations $\lambda = (\delta_0, S_0)(\delta_1, S_1) \cdots (\delta_n, S_n)$ such that $\mathcal{D} \cup \mathcal{C} \models Trans(\delta_i, S_i, \delta_{i+1}, S_{i+1})$, for all $i \in \{0, \dots, n-1\}$. Here, \mathcal{D} is a BAT and \mathcal{C} is the set of axioms characterizing relations $Trans$ and $Final$, together with some extra necessary axioms requiring for encoding programs as terms as described in [De Giacomo et al. 2000]. We say that the execution is *final*, or *terminating*, whenever $\mathcal{D} \cup \mathcal{C} \models Final(\delta_n, S_n)$ applies.

Classical Planning. Planning systems are problem-solving algorithms that operate on explicit representations of states and actions [Nau et al. 2004; Geffner and Bonet 2013]. PDDL [McDermott et al. 1998] 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 experienced 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 an adaptive PMS—which we shall name SmartPM—that is able to not only enact KiPs, but also to *automatically* adapt them 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.

4.1. SmartPM Basic Action Theory

A situation calculus BAT D_{SmartPM} 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 some non-fluent predicates:

- *Service*(srv): srv is a service (i.e., a process participant). The predicate can be specialized into further predicates (e.g., *Actor*(srv) and *Robot*(srv)) describing the specific services' roles;
- *Task*(t): t is a task (e.g., GO or TAKEPHOTO may denote the tasks of navigating or taking pictures);
- *Capability*(c): c is a capability (e.g., *camera* or *extinguisher* may denote the ability to take pictures or extinguish fire);
- *Provides*(srv, c): service srv provides capability c ; and
- *Requires*(t, c): task t requires the capability c .

Observe all these predicates are rigid: their truth values 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

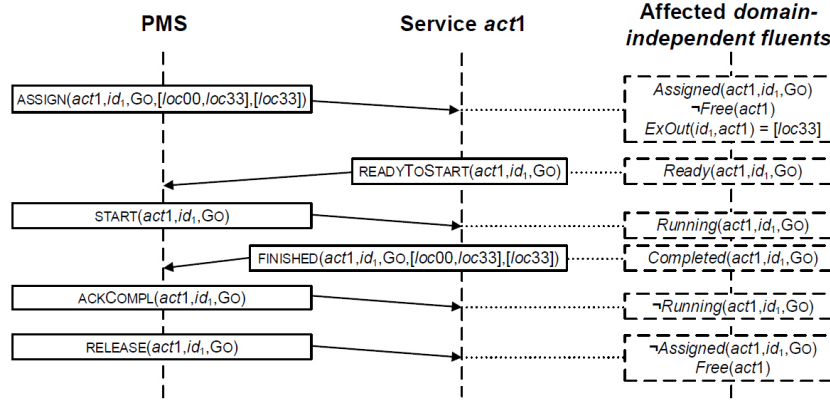


Fig. 3. The protocol for task assignment and execution in SmartPM.

abbreviation.⁶

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

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 external actions arising from services. More concretely, the protocol for the 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 $ASSIGN(srv, id, t, \vec{i}, \vec{o}_e)$, where \vec{i} is an *input data vector* associated to $t:id$ and \vec{o}_e is a vector of *expected* (sensing result) outputs.
- (2) When a service is ready for task execution, it generates the external 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) A specific task finishes execution when either:
 - (a) The service completes the task and hence generates the external action $FINISHED(srv, id, t, \vec{i}, \vec{o}_r)$, with \vec{o}_r representing a vector of *physical* actual outcomes returned by the task execution (we use ϵ to denote the empty output).
 - (b) An exogenous event (e.g., an actor being lost in unexpected location) interfering with the task in question (e.g., actor navigation) arises and hence invalidating the task.

- In both cases, the PMS informs (or acknowledges) the corresponding service of the task completion by means of executing primitive action $ACKCOMPL(srv, id, t)$.
- (5) In either of the above two cases for termination, the PMS updates the properties (i.e., the fluents) to reflect the effects of the task just completed.
 - (6) Finally, the PMS releases the task from the service via action $RELEASE(srv, id, t)$.

Note that we suppose to work in domains in which services, tasks, input and output data vectors are finite. As illustrated in Figure 3, the protocol for task assignment and execution has been applied to an instance of the task GO (specifically, $GO : id_1$)

⁶An abbreviation is a predicate (with situation argument s) defined by means of a formula uniform in s . Abbreviations, unlike fluents, are not directly affected by actions. However, similarly to fluents, their truth value may vary from situation to situation.

which instructs actor *act1* to move from location *loc00* to location *loc33*. The input data vector and the expected output associated to $GO:id_1$ are $\vec{i}=[loc00, loc33]$ and $\vec{o}_e = [loc33]$ respectively⁷. Note that when *act1* launches the FINISHED action (meaning that GO has been completed), the physical output associated to $GO:id_1$ is $\vec{o}_r = [loc33]$, i.e., it is equal to $\vec{o}_e = [loc33]$, that is the final destination expected at design-time. Conversely, the process in Figure 2(b) reflects a wrong execution of the task GO, with a physical output different from the expected one (i.e., $\vec{o}_r \neq \vec{o}_e$).

The above protocol for the life-cycle of tasks is captured by means of a set of *domain-independent* fluents and actions. The relevant successor state axioms (including [Reiter 2001]’s solution to the frame problem) and precondition axioms are as follows:

$$\begin{aligned} \text{Assigned}(srv, id, t, do(a, s)) &\equiv \\ &(\exists \vec{i}, \vec{o}_e) a = \text{ASSIGN}(srv, id, t, \vec{i}, \vec{o}_e) \vee [\text{Assigned}(srv, id, t, s) \wedge a \neq \text{RELEASE}(srv, id, t)]; \\ \text{Ready}(srv, id, t, do(a, s)) &\equiv [a = \text{READYTOSTART}(srv, id, t) \vee \text{Ready}(srv, id, t, s)]; \\ \text{Running}(srv, id, t, do(a, s)) &\equiv \\ &a = \text{START}(srv, id, t) \vee [\text{Running}(srv, id, t, s) \wedge a \neq \text{ACKCOMPL}(srv, id, t)]; \\ \text{Free}(srv, s) &\stackrel{\text{def}}{=} (\nexists t, id) \text{Assigned}(srv, id, t, s). \end{aligned}$$

$$\begin{aligned} \text{Poss}(\text{ASSIGN}(srv, id, t, \vec{i}, \vec{o}_e), s) &\equiv \text{Free}(srv, s) \wedge \text{Capable}(srv, t) \wedge \Phi(srv, t, \vec{i}, \vec{o}_e, s); \\ \text{Poss}(\text{START}(srv, id, t), s) &\equiv \text{Assigned}(srv, id, t, s) \wedge \text{Ready}(srv, id, t, s); \\ \text{Poss}(\text{ACKCOMPL}(srv, id, t), s) &\equiv \text{Running}(srv, id, t, s); \\ \text{Poss}(\text{RELEASE}(srv, id, t), s) &\equiv \neg \text{Running}(srv, id, t, s) \wedge \text{Assigned}(srv, id, t, s). \end{aligned}$$

Most of these axioms are self-explanatory. Observe that a task can be assigned to a service only if this is “free” (i.e., not currently assigned any other task) and capable of performing the task (i.e., provides all capabilities required for the task). In addition, depending on each application, a further $\Phi(srv, t, \vec{i}, \vec{o}_e, s)$ requirement may need to be met. When the service produces a $\text{READYTOSTART}(srv, id, t)$ action for an assigned task, it becomes ready for execution, thus allowing the PMS to instruct actual execution via action $\text{START}(srv, id, t)$. A specific task instance is then considered “running” in the service until completion is acknowledged by the PMS (via action $\text{ACKCOMPL}(srv, id, t)$). Finally, a non-running task can be released from a service (which in term leaves the service “free” for a new assignment).

To record the expected output of task instance $id : t$ when assigned to a service, we define a functional fluent $\text{ExOut}(id, t, s)$, whose successor state axiom is as follows:

$$\begin{aligned} \text{ExOut}(id, t, do(a, s)) = \vec{o} &\equiv \\ &(\exists srv, \vec{i}, \vec{o}_e) a = \text{ASSIGN}(srv, id, t, \vec{i}, \vec{o}_e) \wedge \vec{o} = \vec{o}_e \vee \text{ExOut}(id, t, s) = \vec{o}. \end{aligned}$$

That is, the expected output of a task instance is determined by the assignment step (and never changes). Initial expected outcomes are initialized to “no value” (ϵ) via an axiom $(\forall t, id). \text{ExOut}(id, t, S_0) = \epsilon$ in the set of axioms \mathcal{D}_{S_0} characterizing the initial state of the system.

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 be executed. We call such fluents *data fluents*.

⁷We make use of a Prolog-like notation to represent vectors of inputs/expected outputs/physical outputs as a list of constants enclosed between squared brackets.

They can be seen as features of the world whose value may change from situation to situation. In general, data 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. Importantly, the successors state axioms will follow the following template:

$$F(\vec{x}, do(a, s)) = v \equiv \gamma_F(\vec{x}, a, v, s) \vee [F(\vec{x}, s) = v \wedge (\neg \exists v) \gamma_F(\vec{x}, a, v, s)], \quad (1)$$

where $\gamma_F(\vec{x}, a, v, s)$ states the conditions under which action a executed in situation s will cause data fluent $F(\vec{x}, s)$ to take value v . In the context of our setting, two type of actions will be mentioned in $\gamma_F(\vec{x}, a, v, s)$, namely, (i) actions of the form $\text{FINISHED}(srv, id, T_k, \vec{l}, \vec{o}_r)$ reporting the completion of some task instance $id : T_k$ that would affect the data fluent and (ii) exogenous events that would also affect the value of the data fluent (e.g., an actor got lost in an unexpected location).

Example 4.1. Suppose that, in our emergency scenario, functional data fluent $At(srv, s)$ is used to keep track of the location of service srv in the domain. The fluent is affected by tasks GO and MOVE, but also by exogenous event PUSHED which causes to lose track of the actor's location.

Hence, the successor state axiom for the fluent follows the template (1) above with $\gamma_F(\vec{x}, a, v, s)$ being instantiated as follows:

$$\begin{aligned} \gamma_{At}(srv, a, v, s) \stackrel{\text{def}}{=} & (\exists id, l_s, l_d, t) \\ & (a = \text{FINISHED}(srv, id, t, [l_s, l_d], [v]) \wedge t \in \{\text{GO}, \text{MOVE}\} \wedge (\text{Actor}(srv) \vee \text{Robot}(srv))) \vee \\ & (a = \text{PUSHED}(srv) \wedge \text{Actor}(srv) \wedge v = \text{"lost"}). \end{aligned}$$

In words, actor/robot srv is in location v if srv has just completed the task GO or MOVE whose actual physical outcome result is v , or if human service actor srv got lost and v records the fact that we lost track of its position. Observe that even when a navigation task has just been finished, the new location v of srv may happen to be different to the expected (destination) location l_d . \square

Besides data fluents, an application will generally require further domain-dependent rigid predicates to represent the static properties of a contextual scenario. Such predicates do not change their value during process execution. For example, our case study requires to define a predicate $Neigh(loc1, loc2)$ to expresses the neighborhood property between two locations, and a predicate $Covered(loc)$ to indicate that a location loc is directly covered by the network range provided by the main base network.

Using the core data fluents and static predicates, one can also define helpful abbreviations, such as the following one to capture network connectivity of a service at any point in time.

Example 4.2. The abbreviation $Connected(srv, s)$ denotes that actor srv is within network connectivity range and is defined as follows:

$$\begin{aligned} Connected(srv, s) \stackrel{\text{def}}{=} & \\ & \exists l. At(srv, s) = l \wedge \text{Actor}(srv) \wedge (\text{Covered}(l) \vee \exists r. \text{Robot}(r) \wedge \text{Neigh}(l, At(r, s))). \end{aligned}$$

That is, a human service actor is connected iff it is at a location l that is covered by the main base network, or l is adjacent to a robot providing network connectivity in its surroundings. \square

Now, in KiPs, data fluents and abbreviations will be often used for defining the *pre-conditions* of domain tasks. By doing so, the PMS can reason, at run-time, about the

0:16

Andrea Marrella, Massimo Mecella, Sebastian Sardina

active process instance relative to the current context. For example, while we have given above a generic precondition for assigning tasks to services, one can also incorporate domain-specific restrictions for task assignment.

Example 4.3. The following precondition axiom defines when the PMS can assign a navigation task to a human actor and robot services:

$$\begin{aligned}
& Poss(\text{ASSIGN}(srv, id, \text{GO}, [l_s, l_d], [l_e]), s) \equiv \\
& \quad \text{Free}(srv, s) \wedge \text{Capable}(srv, \text{GO}) \wedge \\
& \quad \text{Actor}(srv) \wedge (\text{At}(srv, s) = l_s) \wedge \text{Connected}(srv, s) \wedge (l_e = l_d); \\
& Poss(\text{ASSIGN}(srv, id, \text{MOVE}, [l_s, l_d], [l_e]), s) \equiv \\
& \quad \text{Free}(srv, s) \wedge \text{Capable}(srv, \text{MOVE}) \wedge \\
& \quad \text{Robot}(srv) \wedge (\text{At}(srv, s) = l_s) \wedge (l_e = l_d) \wedge \\
& \quad \text{EnoughBattery}(\text{BatteryLevel}(srv, s), \text{MoveStep}(l_s, l_d)).
\end{aligned}$$

So, besides the service being available and capable of carrying out the navigation task, it also needs to be an actor located at the source location l_s and currently connected to the network. In addition, the expected outcome of the task instance ought to be the destination location l_d . A similar MOVE action can be used to instruct robots to move between two locations, though it is also required the robot's current battery level is enough to move from source l_s to destination l_d . This latter "constraint" is represented through the abbreviation *EnoughBattery*, which is defined over the fluent *BatteryLevel* (it records the battery charge level of a robot in a specific situation) and the static predicate *MoveStep* (it indicates the cost for a robot to move between two specific locations). \square

Before addressing the issue of how monitoring is modeled in KiPs, we provide the full successor state axiom of another data fluent which is affected by both actions and asynchronous exogenous events.

Example 4.4. Data fluent $Status(l, s)$, which denotes the state of location l at situation s , is affected by three domain tasks that could report the status of locations, as well as two exogenous events: event $ROCKSLIDE(l)$ indicates that a rock slide has collapsed at location l , while event $FIRERISK(l)$ reflects a sudden fire burnt in one of the coaches at location l .

$$\begin{aligned}
& Status(l, do(a, s)) = v \equiv \\
& \quad [(\exists srv, id, t) a = \text{FINISHED}(srv, id, t, [l], [v]) \wedge \\
& \quad \quad t \in \{\text{UPDATESTATUS}, \text{REMOVEDEBRIS}, \text{EXTINGUISHFIRE}\}] \vee \\
& \quad [a = \text{ROCKSLIDE}(l) \wedge v = \text{debris}] \vee \\
& \quad [a = \text{FIRERISK}(l) \wedge v = \text{fire}] \vee \\
& \quad [Status(l, s) = v \wedge \\
& \quad \quad \neg(\exists srv, id, t, v') (a = \text{FINISHED}(srv, id, t, [l], [v']) \wedge \\
& \quad \quad \quad t \in \{\text{UPDATESTATUS}, \text{REMOVEDEBRIS}, \text{EXTINGUISHFIRE}\}) \wedge \\
& \quad \quad a \neq \text{ROCKSLIDE}(l) \wedge a \neq \text{FIRERISK}(l)].
\end{aligned}$$

In words, tasks UPDATESTATUS, EXTINGUISHFIRE, and REMOVEDEBRIS are all meant to report the status of the location they operate on, upon completion. Moreover, rock slide or fire burnt events also change the status of the corresponding location. \square

This concludes the exposition of the first three aspects of a SmartPM action theory, as listed above. We notice that the apparent simplicity, clearness and intuitiveness of the action theory is a direct consequence of having adopted a KR&R-based approach, which provides - as previously stated - the right level of abstraction for modeling knowledge data and the evolution of dynamic situations. In the following section, we will focus on how our framework is able to deal with exception monitoring and management.

4.1.1. *Exception Monitoring.* We now turn our attention to the mechanism for automatically detecting failures/exceptions. In a nutshell, an exception occurs when a task does not produce the expected outcomes or an exogenous event arises.

To monitor and deal with exceptions, we leverage on [De Giacomo et al. 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 or the occurrence of exogenous events, and may require explicit intervention.

The “physical” reality captures the actual value of fluents (and abbreviations) as observed from the system, and is encoded via the data fluents (and abbreviations), as described above (e.g., fluent $At(srv, s)$ and abbreviation $Connected(srv, s)$).

The “expected” reality, in turn, is captured with another set of fluents and abbreviations, one for each one in the physical reality. So, for every data fluent $F(\vec{x}, s)$, a new fluent $F_{exp}(\vec{x}, s)$ (F -expected) is used to represent the value of $F(\vec{x}, s)$ in the “expected” (or “desired”) execution. Technically, if $F(\vec{x}, do(a, s))$ is a relational data fluent with successor state axiom $F(\vec{x}, do(a, s)) \equiv \Psi_F(\vec{x}, a, s)$, then we build $F_{exp}(\vec{x}, s)$ ’s counterpart as follows (the one for functional fluent is built in analogous way):

$$\begin{aligned}
 F_{exp}(\vec{x}, do(a, s)) &\equiv \\
 &[(\exists srv, id, t, \vec{i}, \vec{o}_r) a = FINISHED(srv, id, t, \vec{i}, \vec{o}_r) \supset \\
 &\quad \Psi_F^*(\vec{x}, FINISHED(srv, id, t, \vec{i}, ExOut(id, t, s)), s)] \vee \\
 &[(\exists srv, id, t, \vec{i}) Exog(a) \wedge InteractExogTask_F(a, srv, id, t, \vec{i}) \wedge Running(srv, id, t, s) \supset \\
 &\quad \Psi_F(\vec{x}, FINISHED(srv, id, t, \vec{i}, ExOut(id, t, s)), s)] \vee \\
 &[a \neq FINISHED \supset F_{exp}(\vec{x}, s)].
 \end{aligned} \tag{2}$$

where $\Psi_F^*(\vec{x}, a, s)$ is obtained by replacing every fluent X mentioned 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 condition states that the expected value of the fluent is the value that the fluent would get if all tasks executed so far since the last alignment point end with their expected output (denoted with term $ExOut(id, t, s)$). Basically, when a FINISHED action is invoked, the F ’s successor state axiom is used with the expected outcome in place of the real outcome (i.e., \vec{o}_r is replaced with $ExOut(id, t, s)$).

The second condition is more complex and relates to the occurrence of an external exogenous event affecting the data fluent whose value was expected to be affected by a running task. In a nutshell, it says that if an exogenous event arises (e.g., the fact an actor got lost) that affects the fluent (e.g., actor’s location) which in turn was meant to be affected by a running task (e.g., a navigation task)—that is, $InteractExogTask_F(a, srv, id, t, \vec{i})$ will hold true—then assume that the task has finished with its expected outcome. By doing so, the expected version of a fluent will take the expected value of the fluent as if the running task would have finished successfully. Importantly, it turns out that, when data fluents adhere to the template (1) above, we can define when an exogenous event a interacts with a data fluent F and a task instance as follows (both relational and functional data fluent version are provided):

$$\begin{aligned}
 InteractExogTask_F(a, srv, id, t, \vec{i}) &\stackrel{\text{def}}{=} \\
 &(\exists \vec{x}, a'). (\gamma_F^+(\vec{x}, a, srv, s) \vee \gamma_F^-(\vec{x}, a, srv, s)) \wedge \\
 &\quad a' = FINISHED(srv, id, t, \vec{i}, ExOut(id, t, s)) \wedge (\gamma_F^+(\vec{x}, a', srv, s) \vee \gamma_F^-(\vec{x}, a', srv, s)); \\
 InteractExogTask_F(a, srv, id, t, \vec{i}) &\stackrel{\text{def}}{=} \\
 &(\exists \vec{x}, v, v'). \gamma_F(\vec{x}, a, srv, v, s) \wedge \gamma_F(\vec{x}, FINISHED(srv, id, t, \vec{i}, ExOut(id, t, s)), srv, v', s).
 \end{aligned}$$

0:18

Andrea Marrella, Massimo Mecella, Sebastian Sardina

In words, an exogenous event a interacts with a fluent F and a task instance $id : t$ whenever (a) action a sets the (truth) value of the fluent (first conjunct); and (b) the finishing of task instance $id : t$ with its expected outcome would affect the (truth) value of the fluent F as well.

Example 4.5. Consider again the situation described in Figure 1(e), depicting a rock slide collapsed in location $loc31$ while $act2$ was evaluating the damages in that area (i.e., $act2$ was executing the $UPDATESTATUS(loc31)$ task). Due to the successor state axiom for fluent $Status(x, s)$, exogenous event $ROCKSLIDE(loc31)$ will immediately set the value of data fluent $Status(loc31, s)$ to “debris” (see Example 4.4). However, such a fluent was expected to be eventually affected by current running task $UPDATESTATUS(loc31)$, which is being executed by actor $act2$. That is, the exogenous event *interferes* with the task. To resolve this interaction, the PMS will force the task $UPDATESTATUS(loc31)$ to terminate (see next section). Moreover, the corresponding expected fluent $Status_{exp}(loc31, s)$ shall be set to the expected outcome that would be returned if the task would have not been aborted (see Figure 4). This is exactly what the third condition amounts to. Notice that, as a result, the real value of the fluent and its expected value will typically differ after such exogenous event. \square

We note also that this treatment of exogenous events is coherent with the typical semantics adopted by the majority of PMSs (see [Grefen et al. 2001; Leymann 2001]), where each task is considered as a black box activity, non-interruptible, but possibly *compensable* in case of abortion. The mechanism adopted by SmartPM consists of considering an aborted task as if it has been completely executed. Notably, the solution performs correctly when the control flow of the process is *well-structured*, an assumption we rely on (cf. Section 3).

Finally, the third condition states that for all other cases, the expected fluent keeps the value it had before (i.e., inertia law is applicable).

Similarly, for each abbreviation $A(s) \stackrel{\text{def}}{=} \Psi(s)$, an extra abbreviation $A_{exp}(s)$ is defined as $A_{exp}(s) \stackrel{\text{def}}{=} \Psi^*(s)$, where $\Psi^*(s)$ is obtained by replacing each fluent (or abbreviation) X in $\Psi(s)$ with its expected version X_{exp} .

Therefore, given a physical reality, an expected reality can be seen as the collection of values of the expected versions of data fluents and abbreviations.

Example 4.6. Upon (syntactic and semantic) simplification, the expected version for fluent $At_{exp}(srv, s)$ is as follows:

$$\begin{aligned}
 At_{exp}(srv, do(a, s)) = l \equiv & \\
 & [(\exists id, \vec{i}, t, \vec{o}_r) a = FINISHED(srv, id, t, \vec{i}, \vec{o}_r) \supset \\
 & \quad t \in \{GO, MOVE\} \wedge (Actor(srv) \vee Robot(srv)) \supset l = ExOut(id, t, s)] \vee \\
 & [(\exists id, t, \vec{i}) a = PUSHED(srv) \wedge Actor(srv) \wedge Running(srv, id, t, s) \wedge t \in \{GO, MOVE\} \supset \\
 & \quad l = ExOut(id, t, s)] \vee \\
 & \neg(\exists id, \vec{i}, t, \vec{o}_r) a = FINISHED(srv, id, t, \vec{i}, \vec{o}_r) \wedge \\
 & \quad \neg(\exists id, t) a = PUSHED(srv) \wedge Actor(srv) \wedge Running(srv, id, t, s) \wedge t \in \{GO, MOVE\} \supset \\
 & \quad At_{exp}(srv, s) = l].
 \end{aligned}$$

For abbreviation $Connected(x, s)$, however, we want to force the constraint/expectation of always being connected, and hence we simply take $Connected_{exp}(x, s) \stackrel{\text{def}}{=} \text{true}$. \square

Next, using the data fluents and their expected versions, a misalignment can be recognized and a recovery procedure may be needed. However, it may only be important to check for mismatches among some properties of the world. So, a data fluent (or

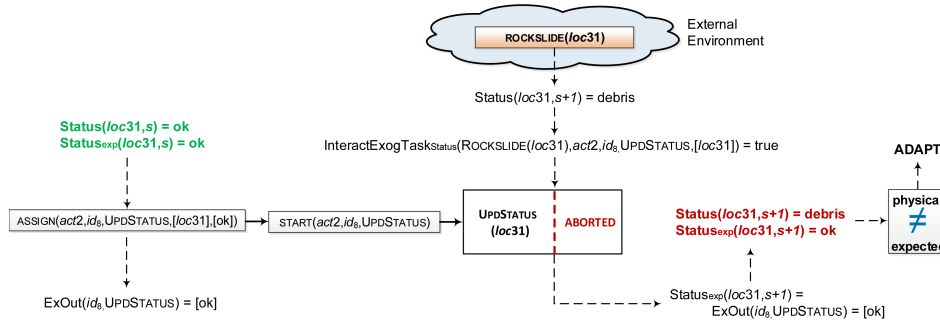


Fig. 4. Task abortion due the occurrence of an exogenous event.

abbreviation) is considered *relevant* by the process designer if its evolution should be monitored during process enactment. We assume then that the designer specifies abbreviation *Misaligned*(*s*) to characterize misalignment situations that would require process adaptation. The general form of such an abbreviation is as follows:

$$\begin{aligned} \text{Misaligned}(s) &\stackrel{\text{def}}{=} \\ &\exists \vec{x}_1. \Phi_{F^1}(\vec{x}_1, s) \supset \neg[F^1(\vec{x}_1, s) \equiv F_{\text{exp}}^1(\vec{x}_1, s)] \vee \\ &\quad \vdots \\ &\exists \vec{x}_n. \Phi_{F^n}(\vec{x}_n, s) \supset \neg[F^n(\vec{x}_n, s) \equiv F_{\text{exp}}^n(\vec{x}_n, s)], \end{aligned}$$

where $F^i(\vec{x}_i, s)$, with $i \in \{1, \dots, n\}$, are all the data fluents and abbreviations used in the SmartPM application. Each condition $\Phi_{F^i}(\vec{x}_i, s)$ states the conditions under which data fluent $F^i(\vec{x}_i, s)$ is relevant and needs to be traced for “misalignment.”

Example 4.7. In our case study, we are interested, among other things, in monitoring the correct location of human actors and their permanent connectivity to the network. Technically, we model that as follows:

$$\begin{aligned} \text{Misaligned}(s) &\stackrel{\text{def}}{=} \\ &\exists x_1. \text{Actor}(x_1) \supset \neg[\text{At}(x_1, s) \equiv \text{At}_{\text{exp}}(x_1, s)] \vee \\ &\exists x_1. \text{Actor}(x_1) \supset \neg[\text{Connected}(x_1, s) \equiv \text{Connected}_{\text{exp}}(x_1, s)] \vee \\ &\exists l_1. \text{Location}(l_1) \supset \neg[\text{Status}(l_1, s) \equiv \text{Status}_{\text{exp}}(l_1, s)] \vee \\ &\quad \vdots \end{aligned}$$

Observe the definition is not concerned about exceptions on the location of robots, for example. \square

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

4.2. SmartPM High-Level Program

A SmartPM application involves the *online execution* (see Section 3) of IndiGolog program $(\text{SmartPM} \parallel \delta_{\text{exog}})$ modeling the concurrent execution of the specific application with special program $\delta_{\text{exog}} = (\pi a. \text{Exog}(a)?; a)^*$ accounting for all potential exogenous events that may arise from the external environment. Algorithm 1 shows a fragment of the IndiGolog program for a SmartPM application. The program, as any high-level program, is meant to be executed relative to a $\mathcal{D}_{\text{SmartPM}}$ BAT as developed above, which shall give meaning to conditions and primitive statements in the program (i.e., ac-

0:20

Andrea Marrella, Massimo Mecella, Sebastian Sardina

tions). We note that the only domain-dependent part in Algorithm 1 is procedure **Process**—all other procedures remain unchanged across applications.

The top-level part of the PMS involves five interrupts running at different priorities, as long as the domain process is yet not finished. The highest three priority programs deal with automated process adaptation; the fourth deals with actual process execution; and the last one forces the system to wait for further changes.

At top priority, the PMS immediately acknowledges (to the corresponding service) the termination of a task (and releases it from the service), of any that happens to be deemed just completed. A running task is deemed complete if it was just reported finished by the service, or an exogenous event has just happened that interacts with some fluent and some instance task. In the latter case, the PMS shall abort the task in question as an exogenous event has changed a property of the world that was related to the outcome of the task. Formally, this is captured with the following auxiliary fluent (which is used in the SmartPM procedure):

$$\begin{aligned} \text{Completed}(srv, id, t, do(a, s)) &\equiv \\ &(\exists \vec{i}, \vec{o}_r) a = \text{FINISHED}(srv, id, t, \vec{i}, \vec{o}_r) \vee \\ &(\exists \vec{i}) a = \text{Exog}(a) \wedge \bigvee_F \text{InteractExogTask}_F(a, srv, id, t, \vec{i}) \wedge \text{Running}(srv, id, t, s) \vee \\ &\text{Completed}(srv, id, t, s). \end{aligned}$$

If no exogenous event-action-task interaction has been identified, then the main procedure checks whether a misalignment has been identified (i.e., fluent *Misaligned* holds). If so, process adaptation is initiated by calling procedure **Adapt** (see subsection 4.3 below). The third interrupt triggers whenever there is a misalignment but the adaptation procedure (in the second priority interrupt) was not able to find a successful plan to repair such misalignment. In that case, the whole execution waits, for some exogenous events that can allow the system to adapt. Though outside the scope of this paper, another possibility in such cases would be to resort to alternative orthogonal adaptation techniques, such as planning from first-principles (as done in [van Beest et al. 2014]) or just require human intervention (see Section 8 for further discussion on this).

Whenever there is no adaptation process in place, the PMS runs the IndiGolog program reflecting the actual KiP (fourth interrupt), by executing procedure **Process**. Recall that the actual KiP (cf. Figure 1(a)) is indeed modelled as yet another IndiGolog program. Managing the life-cycle of a task instance—procedure **ManageExecution**—involves selecting a free and capable service for carrying the task out, assigning the task to the chosen service, and instructing the start of its execution. Note the use of the non-deterministic choice of argument $\pi_{srv}.\delta(srv)$ to select (any) appropriate service.

Finally, at the lowest priority (when the process cannot advance) the PMS just waits for an external action to arrive from one of the services (e.g., a FINISHED action signaling the completion of a running task). We note that, while waiting, the (human) process designer could also manually intervene (for example, by adding new services or updating the capabilities of existing services).

4.3. Process Adaptation via Classical Planning

Possibly 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 and execute actions zero, one, or more times such that abbreviation *Misaligned(s)* becomes false*. 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.

ALGORITHM 1: IndiGolog high-level program for PMS.

```

Proc SmartPM
   $\langle\langle (srv, id, task), \neg Finished \wedge Completed(srv, id, task) \wedge Running(srv, id, task) \rightarrow$ 
    ACKCOMPL( $srv, id, task$ ); RELEASE( $srv, id, task$ )  $\rangle\rangle$ 
   $\langle\langle \neg Finished \wedge Misaligned \rightarrow \mathbf{Adapt} \rangle\rangle$ 
   $\langle\langle \neg Finished \wedge Misaligned \rightarrow \mathbf{WAIT} \rangle\rangle$ 
   $\langle\langle \neg Finished \rightarrow \mathbf{Process}; \mathbf{FINISH} \rangle\rangle$ 
   $\langle\langle \neg Finished \rightarrow \mathbf{WAIT} \rangle\rangle$ .
Proc Adapt
   $\Sigma[(\pi a.a)^*; \neg Misaligned?];$ 
Proc ManagementExecution( $Task, id, Input, ExpOut$ )
   $(\pi srv).$ 
   $(Capable(srv, Task) \wedge Free(srv))?$ ;
  ASSIGN( $srv, id, Task, Input, ExpOut$ );
  START( $srv, id, Task$ );

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

```

Putting it all together, let us formally capture the type of adaptation realized by our approach.

Definition 4.8. Let S be a ground situation term such that $\mathcal{D}_{\text{SmartPM}} \models \text{Misaligned}(S)$.

We say that situation S is *recoverable* if and only if it is the case that $\mathcal{D}_{\text{SmartPM}} \models \exists s'. S \sqsubset s' \wedge \text{executable}(s') \wedge \neg \text{Misaligned}(s')$.

That is, a given situation can be recovered if there is an executable sequence of actions from it that will eventually resolve the misalignment between the physical and expected realities.

So, the following result states that when the execution reaches a misalignment, the **SmartPM** procedure (i) will not execute the main process (encoded in procedure **Process** and running at the fourth priority level); and (ii) will execute the first action of a plan to recover the current situation from misalignment, if such a plan exists (otherwise, the system just waits; see above).

THEOREM 4.9. *Let $(\delta_0, S_0)(\delta_1, S_1) \cdots (\delta_n, S_n)(\delta_{n+1}, S_{n+1})$ be an online execution of IndiGolog program $\delta_0 = (\mathbf{SmartPM})_{\|\delta_{\text{exog}}}$ such that $\mathcal{D}_{\text{SmartPM}} \models \text{Misaligned}(S_n) \wedge \neg \text{Finished}(S_n)$ and if $S_{n+1} = \text{do}(A_{n+1}, S_n)$, $\mathcal{D}_{\text{SmartPM}} \models \neg \text{Exog}(A_{n+1})$. Then:*

- (1) $\delta_n = (\delta_n^1 \gg \delta_n^2 \gg \delta_n^3 \gg \delta_n^4 \gg \delta_n^5)_{\|\delta_{\text{exog}}}$ and $\delta_{n+1} = (\delta_{n+1}^1 \gg \delta_{n+1}^2 \gg \delta_{n+1}^3 \gg \delta_{n+1}^4 \gg \delta_{n+1}^5)_{\|\delta_{\text{exog}}}$;
- (2) $\delta_{n+1}^4 = \delta_n^4$, that is, procedure **Process** representing the main business process has not executed in the last execution step, and $\delta_{n+1}^5 = \delta_n^5$; and

0:22

Andrea Marrella, Massimo Mecella, Sebastian Sardina

- (3) if situation S_n is recoverable, then:
- (a) $\delta_{n+1}^3 = \delta_n^3$, that is, the last execution step is due to the first or second priority program; and
 - (b) if $\delta_{n+1}^1 = \delta_n^1$, then $S_{n+1} = do(A, S_n)$ and $\mathcal{D} \models \exists s. S_{n+1} \sqsubseteq s \wedge Executable(s) \wedge \neg Misaligned(s)$.
- (4) if situation S_n is not recoverable, then either $\delta_{n+1}^1 \neq \delta_n^1$, then $S_{n+1} = do(WAIT, S_n)$.

PROOF. The first point only states the shape of the program at steps n and $n + 1$ and it follows directly from the semantics of the prioritized concurrency \gg and the fact that programs at each priority level are interrupts.

The second point follows from the fact that because $\mathcal{D}_{SmartPM} \models \neg Finished(S_n) \wedge Misaligned(S_n)$, the third priority process is always enabled and ready to execute; hence, the fourth and fifth priority processes will remain still.

For the third point, suppose that S_n is indeed a recoverable situation, as per Definition 4.8. Now, because by assumption we are only interested in steps generated by the **SmartPM** process (and not the exogenous events arising), the last execution step has to be due to one of the first three priority processes in **SmartPM**, that is, δ_n^1 , δ_n^2 , or δ_n^3 . However, because we know that S_n is recoverable, there exists a sequence of actions that will eventually make fluent $Misaligned(s)$ false, and therefore the **Adapt** procedure running at the second priority level is enabled and will generate an action if executed. All this implies that the third priority process will remain still and that if the last execution step was not due to the first priority process (to respond to a task that has just completed), then S_{n+1} is due to procedure **Adapt** and corresponds to an action A that is the first action in a sequence of executable actions leading to a future situation in which fluent $Misaligned(s)$ is false. That is, the third item follows.

Finally, if however S_n is not recoverable, then procedure **Adapt** can not execute any step (i.e., it is blocked); this means that the third priority process is enabled and a **WAIT** action is generated by the **SmartPM** procedure. \square

Importantly, the above result also makes explicit the limits of our adaptation framework: if there is no plan able to resolve the existing mismatch, the system just waits (fourth point) and other orthogonal adaptation techniques would need to be used, see discussion in Section 8.

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. The fact is that the search operator provided by IndiGolog amounts to a generic and incremental (i.e., done at every step) search; as a result, direct implementations are not able to cope with even extremely easy adaptation tasks.

But, 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* type of program that turns out to encode a (classical) planning problem. So, leveraging on 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. Specifically, we used the LPG-td system [Gerevini et al. 2004], one of the many state-of-the-art planning systems available. The basic search scheme of LPG-td is inspired by Walksat [Selman et al. 1994], an efficient procedure for solving SAT-problems; as expected, it outperforms the blind search operator by several orders of magnitudes (cf. Section 6). 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.⁸ First of all, given a BAT D_{SmartPM} for a KiP application as above (Section 4.1), the corresponding PDDL planning domain is built (and stored) offline. Because we are not concerned with the external 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 (expected) 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 modeling the `GO` task:⁹

```
(:action go
:parameters (?srv - actor ?from - location ?to - location)
:precondition
  (and (provides ?srv movement) (free ?srv)
        (at ?srv ?from) (connected ?srv))
:effect (and (not (at ?srv ?from)) (at ?srv ?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 physical reality is encoded as the planning problem *initial state* (as the set of fluents that are true) and the expected reality is encoded as the problem *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, such a 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).

5. SYSTEM ARCHITECTURE

Our approach to integration of process execution, knowledge representation and planning to provide automated adaptation features at run-time is concretely supported by the SmartPM system, which relies on three main architectural layers as shown in Figure 5.

The **Presentation Layer** has a twofold purpose. On one hand, it provides a GUI-based tool called SmartPM Definition Tool. The SmartPM Definition Tool supports the process design activity by providing (i) a wizard-based GUI that assists the process designer in the definition of the process knowledge, and (ii) a graphical editor to design the control flow of a KiP using a subset of the BPMN 2.0 notation [BPML.org and OMG 2011]. The SmartPM Definition Tool has been developed as a standard Java application, using the Java SE 7 Platform, and the JGraphX open source graphical library.¹⁰ The use of the SmartPM Definition Tool allows a human process designer to

⁸The integration of PDDL planning with situation calculus and Golog-like languages has been already analysed in several research works, such as [Claßen et al. 2007b; Claßen et al. 2007a; Fritz et al. 2008].

⁹In PDDL, the variables are distinguished by a '?' character at front, for example `?x1` represents a variable. The dash '-' is used to assign types to the variables. Notice that preconditions and effects of a given action are bundled together, while in the situation calculus they are spread over multiple axioms.

¹⁰<http://www.jgraph.com/>

0:24

Andrea Marrella, Massimo Mecella, Sebastian Sardina

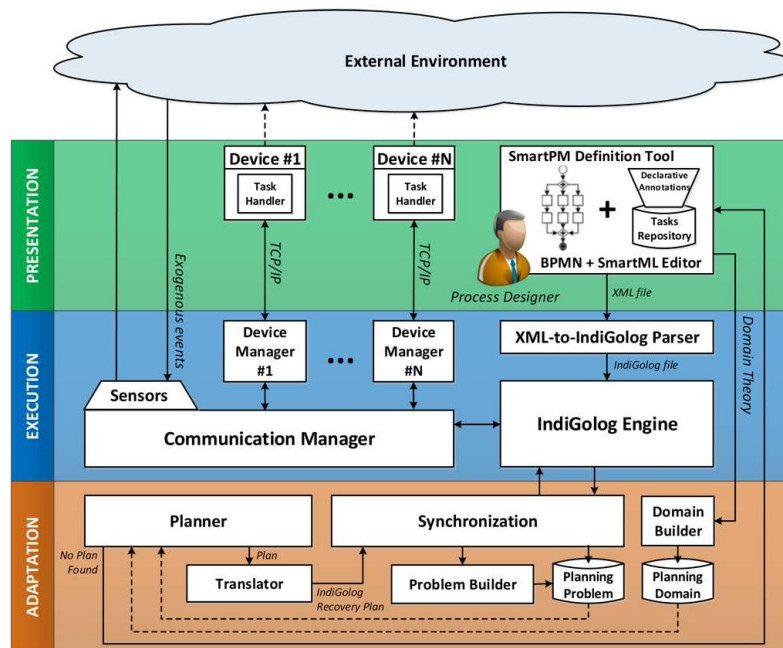


Fig. 5. The architecture of SmartPM.

enter knowledge on processes without being expert of the internal working of the AI tools involved in the system. This means that a process designer that wants to interact with SmartPM is not required to model the process knowledge through complex languages such as situation calculus and IndiGolog. We use such languages for reasoning on the process under execution and for generating recovery procedures at run-time.

From the process designer perspective, process knowledge is represented as a *domain theory* through SmartML, that is the SmartPM Modeling Language.¹¹ SmartML is a declarative language used for representing tasks, services and all the contextual information of the domain of concern. The domain theory involves capturing a set T of n task definitions and supporting information, such as the people/agents that may be involved in performing the process (roles or participants), the data and so forth. Data are represented through some ground *atomic terms* that range over a set of tuples (i.e., unordered sets of zero or more attributes) of *data objects*, defined over some *data types*. In short, a data object depicts an entity of interest (e.g., a location, a specific process participant, etc.). Tasks are collected in a specific repository, and each task $t_i \in T$ (with $i \in 1..n$) is described in terms of its preconditions Pre_i and effects Eff_i . Preconditions are logical constraints defined as a conjunction of atomic terms, and they can be used to constrain the task assignment and must be satisfied before the task is applied, while effects establish the outcome of a task after its execution.

Starting from a SmartML specification of the contextual information in which the process is meant to run, the process designer can define the control flow of a KiP through the BPMN graphical editor provided by the SmartPM Definition Tool. Note that the set of tasks composing the control flow must be selected from the repository of available tasks defined according to SmartML. A screenshot of the workspace of the SmartPM Definition Tool is shown in Figure 6. The main elements of the workspace are

¹¹We describe the complete syntax of SmartML in the appendix.

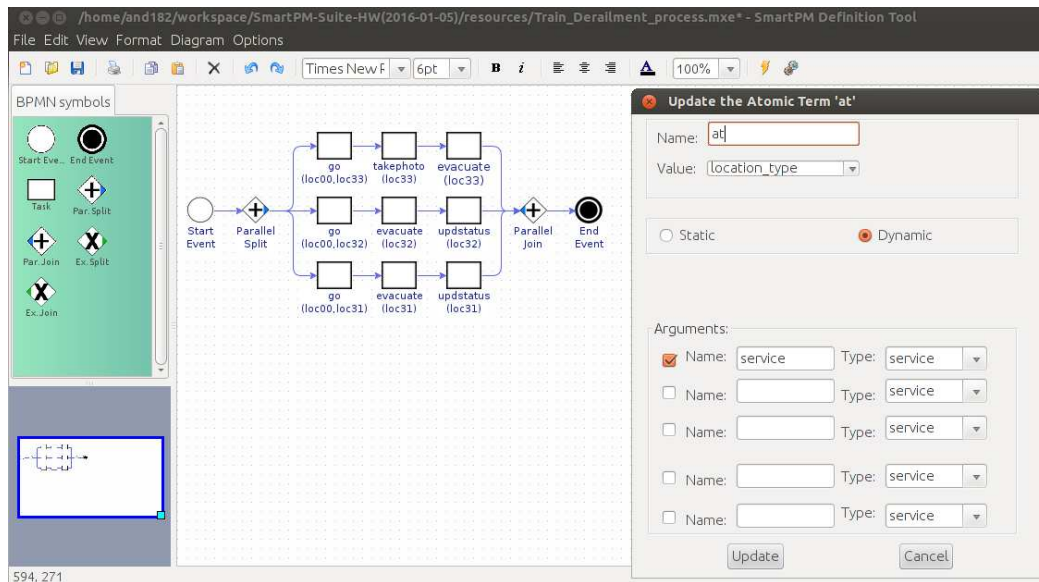


Fig. 6. A screenshot of the SmartPM Definition Tool.

a *modeling canvas* and a *context menu*. The modeling canvas is where BPMN process elements (dragged and dropped from a *process elements panel* located at the left of the workspace) are placed to create and modify the control flow of a process specification. The context menu is a pop-up menu that appears upon a right-click mouse operation on any blank area of the modeling canvas. It presents several modeling options, but the main ones allow to (i) create a new domain theory through a wizard-based GUI, (ii) instantiate an initial description of the execution context and (iii) run the process. The outcome of the process design activity is a complete XML-encoded process specification that is passed to the Execution Layer.

On the other side, the Presentation Layer includes every real world device that may interact with the Execution Layer. In order to manage such an interaction, there is the need to installing and configuring a *Task Handler* module on top of every device. The Task Handler is an interactive GUI-based software application developed in Java that - during process execution - supports the visualization of assigned tasks and allows each participant to start task execution and notify task completion by selecting an appropriate outcome. Figure 7 shows a screenshot of the Task Handler component provided by SmartPM. In the figure, it is described how the data fluent $At(act1)$ can be used to represent the effect of the task GO. We show that the output value for $At(act1)$ (in the example 'loc03', different from the task's expected outcome, that is 'loc33') can be produced by a sensor (i.e., a GPS device) supporting the Task Handler.

The **Execution Layer** is in charge of managing and coordinating the execution of KiPs. It performs task assignment and manages information about services involved in process execution, tasks to be completed and data modified during process execution.

A KiP built through the BPMN notation and annotated with SmartML is taken as input from the *XML-to-IndiGolog Parser* component, which translates this specification in situation calculus and IndiGolog readable formats, in order to make the process executable by the IndiGolog engine. Basically, the XML-to-IndiGolog Parser builds a $D_{SmartPM}$ theory and an IndiGolog high-level program as shown in the previous section.

0:26

Andrea Marrella, Massimo Mecella, Sebastian Sardina

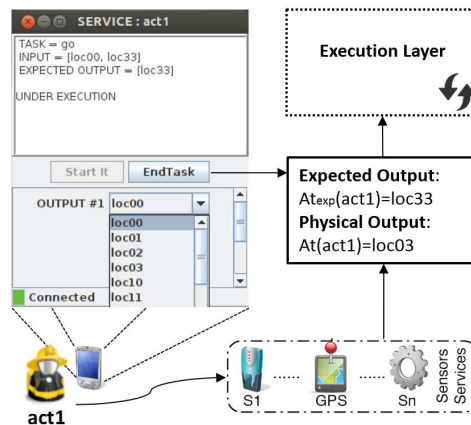


Fig. 7. The task handler of SmartPM.

The *IndiGolog Engine* is a logic-programming implementation of IndiGolog, and allows the incremental execution of high-level IndiGolog programs. It is based on the IndiGolog interpreter¹² developed at University of Toronto (Canada) and RMIT University in Melbourne (Australia). The engine is written in the well-known open source SWI-Prolog environment.¹³ Basically, the IndiGolog engine manages the process routing and decides which tasks are enabled for execution, by taking into account the control flow, the value of predicates/fluents and precondition and effect axioms associated to each task. Before a process starts its execution, the IndiGolog engine builds its physical reality by taking the initial context from the external environment.

The IndiGolog Engine is also in charge of monitoring contextual data in order to identify changes or events which may affect process execution. Specifically, at each execution step - i.e., when the completion of a task or an exogenous event has occurred in situation s - the engine checks if in the new situation $do(a, s)$ (where a is a FINISHED action or an exogenous event) the physical and expected realities are misaligned. If this is the case, the engine (i) stops the main process execution, (ii) collects the physical and expected realities and (iii) sends them to the Synchronization component, that is in charge to decide if process adaptation is required.

Once a task is ready for being assigned, a component named *Communication Manager* assigns it to a proper service that is available (i.e., free from any other task assignment) and that provides all the required capabilities for task execution. Every step of the task life-cycle - ranging from the assignment to the release of a task - requires an interaction between the Communication Manager and the devices. For each real world device, the Communication Manager generates a separate *Device Manager*, which is a software component able to interact with the task handlers deployed on the devices. Each device manager establishes a communications channel with the associated devices by using TCP/IP stream sockets. Such an interaction is mainly intended for notifying a device of actions performed by the IndiGolog engine as well as for notifying the engine of actions executed by the task handlers of the devices. Finally, the Communication Manager provides some ad-hoc sensors for collecting exogenous events coming from the external environment.

The **Adaptation Layer** is in charge of reacting to undesired or unforeseen events which may invalidate process execution. The *Synchronization* component acts as

¹²<https://bitbucket.org/ssardina/indigolog>

¹³<http://www.swi-prolog.org/>

unique entry point for incoming notifications from the Execution Layer, and enforces synchronization between the IndiGolog engine and the Planner component. When it receives from the IndiGolog engine a representation of the two realities, it builds a corresponding planning problem in PDDL (through the *Problem Builder* component), by converting the physical reality into the initial state and the expected reality into the goal of the planning problem (as described in the previous section).

The *Planner* component is invoked when the Synchronization component builds a new planning problem. In addition, the planner needs a specification of the planning domain (that is, a PDDL specification with tasks and predicates). For this purpose, the *Domain Builder* component translates the SmartML specification in a PDDL planning domain readable by the planner. In the SmartPM system, we synthesize our recovery plans through the LPG-td planner [Gerevini et al. 2004] (Local search for Planning Graphs).

In order to invoke the planner, the Synchronization component is required to specify the value of three parameters indicating: (i) the file containing the set of PDDL operators (i.e., the planning domain, provided by the Domain Builder), (ii) the file containing a planning problem (i.e., the initial state and goal of the problem, provided by the Problem Builder) formalized in PDDL and (iii) the running mode, which is either “speed” (for devising sub-optimal solutions, that do not prove any guarantee other than the correctness of the solution) and “quality” (for devising solutions obtained under a pre-specified metric).

Finally, when a plan satisfying the goal is found, it is sent back to a *Translator* component, that converts it in a readable format for the IndiGolog engine and passes it back to the Synchronization component. The Synchronization component combines the faulty process instance δ' with the recovery plan δ_a just found, and builds the adapted process $\delta'' = (\delta_a; \delta')$ that will be executed by the IndiGolog Engine. If the Planner is not able to find any recovery plan for a specific exception, the control is passed back to the process designer, which can intervene manually to solve the exception.

This concludes the complete overview of the SmartPM system. Let us now look at some empirical evaluation of such a system.

6. VALIDATION

In order to investigate the feasibility of the SmartPM approach, we performed some experiments to learn the time amount needed for synthesizing a recovery plan for different adaptation problems. We made our tests by using two different state-of-the-arts planners, LPG-td [Gerevini et al. 2004] and Fast-Downward [Helmert 2006; 2009]. LPG-td is based on a stochastic local search - inspired by Walksat [Selman et al. 1994] - in the space of particular “action graphs” derived from the planning problem specification. We chose LPG-td as (i) it treats most of the features of PDDL 2.2 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 as it does not stop when the first plan is found but continues until a stopping criterion is met. In our experiments, the criterion for plan quality was set to minimal plan length (i.e., as the smallest number of actions needed for reaching a goal state).

Conversely, Fast Downward is a progression planner that uses hierarchical decompositions of planning tasks for computing its heuristic function, called the *causal graph heuristic*, which approximates goal distances by solving a hierarchy of “local” planning problems. To produce quality plans, Fast Downward uses a best-first search in first iteration to find a plan and a weighted A* search to iteratively decreasing weights of

0:28

Andrea Marrella, Massimo Mecella, Sebastian Sardina

Table I. Time performances of LPG-TD.

L-RP	#I	AT-SOS-i7	AT-SOS-DUO	AL-SOS	AT-QS-i7	AT-QS-DUO
1	29	0.468	1.982	2	1.039	4.843
2	36	0.472	1.998	3	8.356	18.376
3	32	0.539	2.187	5	27.976	47.987
4	25	0.545	2.313	6	42.767	73.012
5	21	0.547	2.314	8	51.023	86.076
6	17	0.551	2.467	9	67.614	91.160
7	13	0.551	2.617	11	81.406	131.820
8	12	0.591	2.742	13	93.741	132.681
9	9	0.593	2.865	14	94.777	179.772
10	6	0.635	3.032	14	143.540	280.981

Table II. Time performances of Fast Downward.

L-RP	#I	AT-SOS-i7	AT-SOS-DUO	AL-SOS	AT-QS-i7	AT-QS-DUO
1	29	1.026	5.034	1	1.172	5.588
2	36	1.067	5.094	2	1.199	5.860
3	32	1.109	5.118	3	1.282	5.989
4	25	1.116	5.127	5	1.328	6.203
5	21	1.123	5.208	6	2.094	8.459
6	17	1.125	5.243	7	5.709	21.875
7	13	1.131	5.443	8	8.616	35.082
8	12	1.139	5.569	9	17.102	69.142
9	9	1.185	5.789	11	28.846	113.636
10	6	1.305	5.943	12	60.975	225.674

Legend. L-RP = length of the recovery plan; #I = number of problem instances; AL-SOS = average length for sub-optimal plan; AT-SOS-i7/AT-SOS-DUO = average time for sub-optimal plan computed with a high/low-end machine; and AT-QS-i7/AT-QS-DUO = average time for quality plan computed with a high/low-end machine.

plans. Notably, if compared with LPG-td, Fast-Downward does not provide any support to the definition of functional numeric fluents.

The experimental setup was performed with the test case shown in our running example. We stored in the task repository 20 different emergency management tasks, annotated with 28 relational predicates and 1 derived predicate. Then, we provided 200 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. We ran both LPG-td and Fast-Downward with the above planning problems and we did our tests by using two different machines: a low-end one based on an Intel U7300 CPU 1.30GHz Dual Core and 4GB RAM, and a high-end one consisting of an Intel Core i7-4770S CPU 3.10GHz Quad Core and 16GB RAM (notably the low-end machine is a representative of a typical computing unit that can be deployed on the field during emergencies, as a rugged embedded computer).

As shown in Tables I and II, the column labeled as *L-RP* (that is “Length of the recovery procedure”) indicates the smallest number of actions needed for devising a plan of a specific length. The column labeled as *AL-SOS* (that is “Average length of a sub-optimal solution”) indicates the average number of actions that compose a sub-optimal solution for a problem of a given complexity. Finally, the columns with the prefix “AT-QS” (which stands for “Average time for a quality solution”) and “AT-SOS” (which stands for “Average time for a sub-optimal solution”) record the computation time needed for finding a quality solution and a sub-optimal solution by using, respectively, the high-end machine (cf. the suffix “i7”) and the low-end machine (cf. the suffix “DUO”).

From the analysis of the results, it is clear that a sub-optimal solution is found in less time than a quality one, but it generally includes more tasks than the ones strictly needed. This is particularly true for the LPG-td planner. For example, as shown in Table I, 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 0.547 seconds with the high-end machine and 2.314 seconds with the low-end machine (with 3 more tasks, on average), and a quality plan (which consists exactly of the 5 tasks needed for the recovery) in 51.023 seconds with the high-end machine and 86.076 seconds with the low-end machine. Conversely, as shown in Table II, Fast-Downward has better performances when computing a quality plan, and lower performances in the synthesis of a sub-optimal plan. In fact, in the case of planning problems requiring a recovery procedure of length 5, Fast-Downward was able to synthesize a sub-optimal plan, on average, in 1.123 seconds with the high-end machine and 5.208 second with the low-end machine, and a quality plan in 2.094 seconds with the high-end machine and 8.459 seconds with the low-end machine. In addition, all sub-optimal solutions provided by Fast-Downward have better “quality” if compared with the ones produced by LPG-td, as they require on average only one or two tasks more than the ones strictly needed.

We notice that business processes, even in dynamic scenarios, are not real-time processes (as the ones, e.g., controlling a nuclear plant) but their duration, and the average duration of tasks, is “human-time”, i.e., in the order of minutes; as an example, cf. [Jennings et al. 2000], just for comparing with a system with similar functionalities applied in an empirical study. Therefore, having automated adaptation solved in seconds, instead of a manual one solved in minutes - and requiring human operators for computing it - is undoubtedly an advantage. The experiments reported above aims at highlighting how performance is made of many dimensions; not only the time of computation have to be considered, but also other parameters like the number of tasks found in the solution (e.g., if each task requires minutes to be performed by a human actor, maybe it is preferable to wait more time - more seconds - in computing a shorter solution). We have created a framework for adaptation in which users can easily change the planning techniques, on the basis of their features, as we have formalized through KR&R the semantics of processes and of adaptation. We have shown we can decrease the time of computation to a few seconds (milliseconds on powerful workstations) by using different planners, maybe losing some expressive power or features (e.g., Fast-Downward does not provide any support to the definition of functional numeric fluents), depending on the specific application domain. In any case, the techniques integrated in our approach are feasible in real applications, being at least an order of magnitude faster than having human intervention. In future work, we would like to conduct an empirical study on these issues (length of tasks, how much time a human perform an adaptation with respect to a software system, and with which quality, etc.).

6.1. 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 instances if some failure 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

0:30

Andrea Marrella, Massimo Mecella, Sebastian Sardina

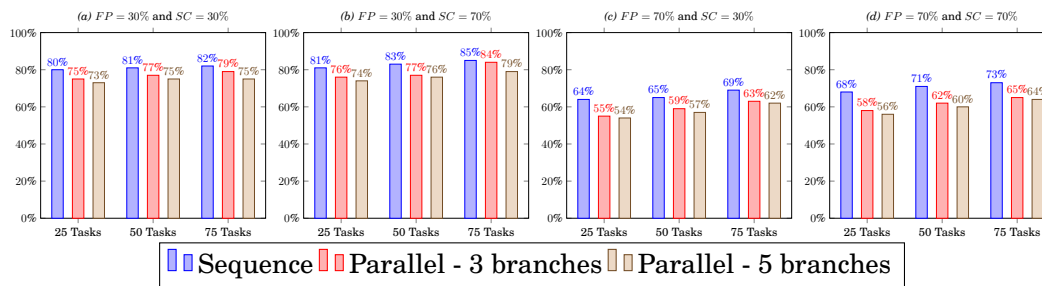


Fig. 8. Analysis of the SmartPM effectiveness. The x-axis states the size of the tasks repository, while the y-axis indicates the effectiveness in executing process models with a specific structure of the control flow.

corresponding D_{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. For example, if a process model is composed by 10 tasks, and the percentage of failures is equal to 70%, we have that 7 tasks of its running process instance will complete with some physical outcome different from the one expected, thus requiring the process to be adapted.
- *Percentage of capabilities coverage* (30% or 70%), that affects the ability of each available service to execute the tasks stored in the repository. During the simulation, we assumed that each task was associated with a unique capability. For example, if the tasks repository stores 75 tasks and the percentage of capabilities coverage is equal to 30%, each available service is able to execute at least 22 tasks 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 8. 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 8(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 3 parallel branches (79%) and in 5 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 F.P. increases, the effectiveness of the SmartPM system decreases; (ii) when the S.C. 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; (iii) to have a large tasks repository helps to increase the effectiveness of the SmartPM system, since the planner has a greater choice when builds a recovery procedure.

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%.

7. RELATED WORK

The issue of having software systems automatically and autonomously adapt to changing conditions has been addressed in the last years under the *autonomic* and *self-healing systems* literature [Ghosh et al. 2007; Psailer and Dustdar 2011]. In such research contexts, SmartPM can be considered as a specific kind of self-healing system addressing the management of processes through AI-based techniques. In particular, according to what presented in [Psailer and Dustdar 2011], its main peculiarities are the use of a context- and data-aware process engine and the ability to reason over dynamic contexts, for which the adopted KR&R techniques - as previously discussed - are particularly suitable.

A research area related to process adaptation is the one of (automatic) Web service composition [Berardi et al. 2003; Pistore et al. 2005a; 2005b; Agarwal et al. 2005; De Giacomo et al. 2014]. Notably, most of the approaches to Web service composition adopt planning-based techniques, as the SmartPM approach does. However, there exists a relevant difference in *what* SmartPM and Web service composition techniques synthesize through planning. In Web service composition, given a set of available services and a target service, the challenge is to synthesize a possible composition (a.k.a. *orchestration*) of the available services such that the target one can be obtained. Depending on the approach used, the target can be a goal to be reached or a business process to be reproduced. Conversely, in the SmartPM approach, the aim is to synthesize a recovery process that *repairs* the original one in a specific situation of the world by using a set of tasks stored in a tasks repository. Without any doubt, tasks and services are similar concepts, as well as a goal/target service is analogous to a situation to be recovered in SmartPM. However, on the one hand, in Web service composition the most important fact is to preserve service behaviors (which are in general quite rich, often modeled as transition systems). On the other hand, in process adaptation (and, in particular, in SmartPM), the main issue is to preserve as much as possible the original process, considering also that process tasks are atomic and do not present rich behaviors to be preserved.

After having provided a general view of the relationships among process adaptation and other relevant area, in the following we describe the state-of-the-art approaches to process adaptation considering to what extent users are involved in the process of defining exception conditions and handling policies (as summarized in Figure 9), which directly influences the degree of automation provided in the exception resolution and

0:32

Andrea Marrella, Massimo Mecella, Sebastian Sardina

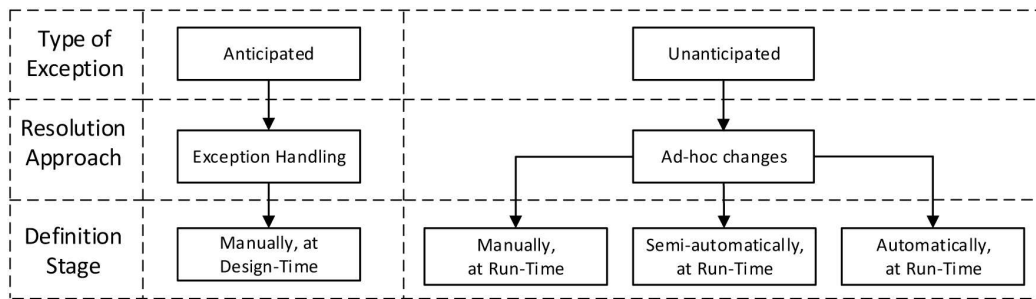


Fig. 9. Exception handling and process adaptation approaches.

process adaptation stages. Specifically, we first outline traditional exception handling techniques used to deal with anticipated exceptions (Section 7.1). Then, we review the existing approaches allowing ad-hoc process changes for adapting running process instances in case of unanticipated exceptions (Section 7.2). Finally, we analyze a number of techniques from the field of AI that have been applied to BPM with the aim of increasing the degree of automated process adaptation at run-time (Section 7.3).

7.1. Exception Handling

Initial research efforts addressing the need for exception handling in PMSs can be traced back to the late nineties and early two thousands [Casati et al. 1999; Casati and Cugola 2001; Eder and Liebhart 1995; 1996; Hagen and Alonso 2000; Klein and Dellarocas 2000; Luo et al. 2000]. Although possible sources of anticipated exceptions are different (as outlined in [Casati and Cugola 2001; Eder and Liebhart 1995], they can be reconducted to activity failures, deadline expirations, resource unavailabilities, constraint violations and external events) and go beyond technical failures, not surprisingly exception handling approaches in PMSs trace and resemble exception handling mechanisms in programming languages.

Abstracting from the specific techniques and implementations, a common behavioral pattern can be identified. At design-time, the process designer identifies possible exceptions that may occur, defines exception triggering events and conditions, and specifies exception handlers associated with the predefined process model. Exception handlers can be defined for single activities, for selected process regions (including multiple activities), or for the overall process (as in the case of a try block in programming languages). The main process logic is thus clearly separated from the exception handling logic. During process execution, timers, messages, errors, constraint violations and other events might interrupt the process flow: the exception is detected and thrown. The run-time environment checks for the availability of a suitable exception handler, which is then invoked to catch the exception (as in the case of a catch block). Typically, the process (or sub-parts of it) is interrupted and the flow of control passes to the exception handler. The handler defines specific activities to be performed to recover from the exception, so that process execution can be possibly resumed.

Typical strategies applied when defining exception handlers for anticipated exceptions have been systematized in the form of *exception handling patterns* [Russell et al. 2006; Lerner et al. 2010; Reichert and Weber 2012]. When for a given exception no explicit handling logic is defined or the handler is not able to resolve the issue, a process participant may be notified and involved in the definition of corrective actions.

As extensively discussed in [Adams 2007], exception handling capabilities provided by academic prototypes and commercial process management systems can be reconducted to the abstract framework introduced before. The different approaches vary in

the exception types that can be handled and in the way they support the definition and selection of exception handlers, which can be completely predefined, contextually selected from a repository or instantiated from templates.

Several exception detection and handler activation techniques [Casati et al. 1999; Hagen and Alonso 2000; Chiu et al. 2000] adopt a rule-based approach, typically relying on some form of Event-Condition-Action (ECA) rules. ECA rules have the form “on *event* if *condition* do *action*” and specify to execute the *action* (i.e., the exception handler) automatically when the *event* happens (i.e., when the exception is caught), provided the a specific *condition* holds. ECA rules represent a good way for separating the graphical representation of the process with the “exception handling flow”.

A similar principle has been applied in YAWL [ter Hofstede et al. 2009], where for each exception that can be anticipated, it is possible to define an exception handling process, named *exlet*, which includes a number of exception handling primitives (for removing, suspending, continuing, completing, failing and restarting a workitem/case) and one or more compensatory processes in the form of *worklets* (i.e., self-contained YAWL specifications executed as a replacement for a workitem or as compensatory processes). Exlets are linked to specifications by defining specific rules (through the Rules Editor graphical tool), in the shape of *Ripple Down Rules* specified as “if condition then conclusion”, where the condition defines the exception triggering condition and the conclusion defines the exlet.

7.2. Ad Hoc Process Change

Even though the handling of anticipated exceptions is fundamental for every PMS, the latter also needs to be able to deal with unanticipated exceptions. Research efforts dealing with unanticipated exceptions have established the area of *adaptive process management* [Weske 2001; Reichert and Weber 2012]. While the introduction of exception handling techniques for anticipated exceptions increases process flexibility and adaptation capabilities, a different approach is required for handling unanticipated exceptions and deviations occurring at run-time. The handling of unanticipated exceptions does not assume the availability of predefined exception handlers and relies on the possibility of performing ad hoc changes over process instances at run-time. The need to perform complex behavioral changes over a process instance requires *structural adaptation* of the corresponding process model. Structural adaptations over the model then lead to adaptations of the process instance state.

As in the case of exception handling, structural adaptation techniques have been systematized through the identification and definition of adaptation patterns [Weber et al. 2008]. At a low-level of abstraction, structural model adaptations can be performed by applying change primitives such as adding/removing nodes, routing elements, edges and other process elements. At a higher level of abstraction, change operations provide a set of adaptation patterns to perform model adaptations, such as adding, deleting, moving or replacing activities or process fragments. A single change operation corresponds to the application of multiple change primitives, hiding the complexity of the model editing task. Adaptation patterns are not limited to the control flow perspective and also cover the other process perspectives to perform changes, for example, at the level of the data flow schema or on process resources. In addition, change operations performed for one perspective (e.g., control flow) may affect the other perspectives (e.g., the data flow) as well, resulting in so-called secondary changes. As a fundamental challenge and requirement, ad hoc changes must preserve the correctness of the process model and the executability and continuation of the process instance [Rinderle et al. 2004].

While a good level of support can be provided to ensure correctness and compliance when high-level change operations are performed, the degree of automation in

performing these changes is generally limited. In adaptive process management environments, ad hoc changes are often manually performed by experienced users: process execution is suspended and the model and state of the affected instance are adapted by relying on the capabilities of the modeling environment.

In an attempt to increase the level of user support, semi-automated approaches have been proposed [Rinderle et al. 2005]. They aim at storing and exploit available knowledge about previously performed changes, so that users can retrieve and apply it when adapting a process. Knowledge retrieval and reuse require to establish a link between performed changes and the application context, including the occurred exception and the process state. Contextual information allows in turn to identify similarities between the current exceptional situation and previous cases. The available knowledge on how similar cases were handled in the past is used to assist the users, provide recommendations and suggest possible changes to be applied. Such an approach has been concretely put into practice using case-based reasoning techniques [Weber et al. 2004; Minor et al. 2014].

Strong support for adaptive process management and exception handling is provided by the ADEPT system and its evolutions [Reichert and Dadam 1998; Reichert et al. 2003; Reichert et al. 2005; Lanz et al. 2011]. ADEPTflex offers modeling capabilities to explicitly define pre-specified exceptions, and supports changes of process instances to enable different kinds of ad-hoc deviations from the pre-modeled process schemas in order to deal with run-time exceptions. These features have been extended and improved in ADEPT2, which provides full support for the structural process change patterns defined in [Weber et al. 2008], and in ProCycle, which combines ADEPT2 with conversational case-based reasoning (CCBR) methodologies. On the basis of the ADEPT technology, the AristaFlow BPM Suite was developed, with the aim of transferring process flexibility and adaptation concepts into an industrial-strength PMS. Similarly, the workflow management system AgentWork [Müller et al. 2004] relies on ADEPTflex and exploits a temporal ECA rule model to automatically detect logical failures and enable both reactive and predictive process adaptation of control- and data-flow elements. Here, exception handling is limited to single tasks failures, and the possibility exists for conflicting rules to generate incompatible actions, which requires manual intervention and resolution.

If compared with traditional exception handling approaches (cf. Section 7.1), we notice that adaptive PMSs deal with unanticipated exceptions by automatically deriving the try block as the situation in which the PMS does not adequately reflect the real-world process anymore. As a consequence, one or several process instances have to be adapted with ad hoc process changes, and the catch block should include those recovery procedures required for realigning the computerized processes with the real-world ones. The fact is that the common strategy used by the adaptive PMSs is to *manually* or *semi-automatically* define at run-time the catch block. However, in dynamic and knowledge-intensive working environments, analyzing and defining these adaptations “manually” becomes time-demanding and error-prone. Indeed, the designer should have a global vision of the application and its context to define appropriate recovery actions, which becomes complicated when the number of relevant context features and their interleaving increases. Conversely, our SmartPM approach is able to *automatically synthesizing at run-time* the catch block, without the need of any manual intervention at run-time, and increases the level of automated process adaptation if compared with the existing state-of-the-art techniques.

7.3. AI-based Process Adaptation

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 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 anticipated 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. Similarly, the approach presented in [R-Moreno and Kearney 2002] highlights the improvements that a legacy workflow application can gain by incorporating planning techniques into its day-to-day operation. Finally, the works [Marrella and Lespérance 2013a; 2013b] investigate how to automatically synthesize process models via partial-order planning techniques starting from an incomplete description of the process domain.

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 redefine 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.

In the work [Bucchiarone et al. 2011] the authors propose a goal-driven approach for service-based applications to automatically adapt business processes to run-time context changes. Process models include service annotations describing how services contribute to the intended goal, and business policies over domain elements. Contextual properties are modeled as state transition systems capturing possible values and possible evolutions in the case of precondition violations or external events. Process and context evolution are continuously monitored and context changes that prevent goal achievement are managed through an adaptation mechanism based on service composition via automated planning techniques. However, this work requires that the process designer explicitly defines the policies for detecting the exceptions at design-time. Conversely, in SmartPM the recovery procedure is synthesized at run-time, without the need to define any recovery policy at design-time.

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 unanticipated or wrong business outcomes. To overcome this limitation, the work [van Beest et al. 2014] 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

0:36

Andrea Marrella, Massimo Mecella, Sebastian Sardina

inconsistencies generated from the interference, which are automatically synthesised through a domain independent planner based on CSP 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.

8. DISCUSSION AND CONCLUSION

This paper has been devoted to define a general approach, a concrete framework and a prototype PMS, called SmartPM, for automated adaptation of KiPs. 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 in case of unanticipated exceptions.

Existing approaches dealing with unanticipated exceptions typically rely on the involvement of process participants at run-time, so that authorized users are allowed to manually perform structural process model adaptation and ad-hoc changes at the instance level. However, KiPs demand a more flexible approach recognizing the fact that in real-world environments process models quickly become outdated and hence require closer interweaving of modeling and execution. To this end, the adaptation mechanism provided by SmartPM 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).

From a general perspective, our planning-based automated exception handling approach should be considered as complementary with respect to existing techniques, acting as a "bridge" between approaches dealing with anticipated exceptions and approaches dealing with unanticipated exceptions. When an exception is detected, the run-time engine may first check the availability of a predefined exception handler, and if no handler was defined it can rely on an automated synthesis of the recovery process. In the case that our planning-based approach fails in synthesizing a suitable handler (or an handler is generated but its execution does not solve the exception), other adaptation techniques need to be used. For example, if the running process provides a well-defined intended goal associated to its execution, we could resort to the van Beest's work [van Beest et al. 2014] and do planning from first-principle to achieve such a goal. Conversely, if no intended goal is associated to the process, a human participant can be involved, leaving her/him the task of manually adapting the process instance. However, the fact that the SmartPM approach relies on well founded KR&R formalisms opens the door for many advanced reasoning tasks upon failure and amounts to very promising future work, e.g., to investigate what parts of the process can not be repaired or abduce what has gone wrong in the past, in order to assist the user in the manual definition of the recovery plan.

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 small/medium-sized real-world problems as used in practice. The empirical experiments reported in Tables I and II confirm the feasibility of the adopted planning-based approach from the timing performance perspective. On the other hand, classical planning imposes some restrictions for addressing more expressive problems, including incomplete information, preferences and multiple task effects. The adoption of classical planning has enforced the adoption of specific requirements that frame the scope of applicability of the approach, which basically relies on the following assumptions:

- process structure can be completely captured in a procedural predefined process model that explicitly defines the tasks and their execution constraints;
- process execution context can be fully captured as part of the process model, i.e., complete information about a fully observable domain is available;
- domain objects and contextual properties representing the state of the world can be reconducted to a finite set of finite-domain variables; and
- process tasks can be completely specified in term of I/O data elements, preconditions and deterministic effects.

In addition to the full observability assumption, the SmartPM approach relies on a high degree of *controllability* over the environment: when process execution deviates from the prespecified expected behavior (i.e., the physical reality deviates from the expected one), it should be possible to synthesize a recovery process whose execution modifies the environment (as reflected in the physical reality) so that the process instance can progress as expected, according to the prespecified model (basically, the physical reality is reconducted to the expected reality). When the operational environment and process state cannot be reconducted to their expected representation, we are back to the case where a process cannot be recovered to progress according to the predefined model, and it is the process itself that has to be (manually or semi-automatically) adapted to the changed environment. These assumptions result from the need of balancing between modeling complexity and expressive power, and the practical requirements that allow to exploit classical planning tools.

Future work will include an extension of our approach to “stress” the above assumptions by making the approach applicable to less-controllable domains, such as *smart spaces*, with the purpose to maintain the planning process very responsive. In such domains, the current widespread availability of wireless network technology for mass consumption has triggered the appearance of plenty of wireless and/or mobile devices providing applications able to enhance the visitors’ experience in cultural sites. The “pre-fixed” and static visits of physical spaces have been turned into interactive dynamic experiences customized to the human visitors’ behaviours and needs. To make SmartPM applicable to such domains, we aim at turning the centralized control provided by SmartPM (in which the reasoning is performed by a single entity, which subsequently instructs the process participants what to do) into a decentralized control, in which each participant will be provided with her/his mobile device with the SmartPM system installed into it. The challenge is to provide each SmartPM system with the ability to adapt the single processes of individual process participants by considering not only the local knowledge collected by the single participant, but also the knowledge produced by the other visitors of the smart space and the global knowledge provided by the smart space as a whole (e.g., the knowledge produced by the sensors installed in the smart space).

We also notice that, even if the SmartPM approach is able to adapt a process instance at run-time, it does not allow neither to support hierarchical processes nor to evolve the original process model on the basis of exceptions captured. Therefore, a second future direction of this work is to provide support for executing hierarchical processes, with high-level processes achieving more general goals that can invoke simpler processes to achieve some of their subgoals. We argue that agent-technology (for example, BDI [Rao and Georgeff 1995], which stands for “believe-desire-intensions”) and hierarchical planners [Nau et al. 2003] can provide promising approaches and methods to address this challenge. In addition, a third main future work concerns to avoid to consider all deviations from the process as errors, but as a natural and valuable part of the work activity, which provides the opportunity for learning and thus evolving the process model for future instantiations.

0:38

Andrea Marrella, Massimo Mecella, Sebastian Sardina

The current prototype of SmartPM is developed to be effectively used by process designers and practitioners.¹⁴ 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. Although the need to explicitly model process execution context and annotate tasks with preconditions and effects may require some extra modeling effort at design-time (also considering that traditional process modeling efforts are often mainly directed to the sole control flow perspective), the overhead is compensated at run-time by the possibility of automating exception handling procedures. While, in general, such modeling effort may seem significant, in practice it is comparable to the effort needed to encode the adaptation logic using alternative methodologies like happens, for example, in rule-based approaches.

A further future work will 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

Andrea Marrella and Massimo Mecella have been partly supported over the years by the following projects: EU FP6 WORKPAD, EU FP7 SM4All, Italian Sapienza grant TESTMED, Italian Sapienza grant SUPER, Italian Sapienza award SPIRITLETS, Italian project Social Museum e Smart Tourism (CTN01.00034.23154), Italian project NEPTIS (PON03PE.00214.3), Italian project RoMA - Resilience of Metropolitan Areas (SCN.00064). Sebastian Sardina acknowledges the support from a Sapienza 2014 Visiting Grant.

The authors would like to thank the many persons involved over the years in the SmartPM conception and development, namely Giuseppe De Giacomo, Massimiliano de Leoni, Patris Halapuu, Arthur H.M. ter Hofstede, Alessandro Russo, Paola Tucceri and Stefano Valentini.

Finally, we would like to thank the anonymous reviewers for their many suggestions that helped us to greatly improve the quality of this article.

REFERENCES

- Michael James Adams. 2007. *Facilitating dynamic flexibility and exception handling for workflows*. Ph.D. Dissertation. Queensland University of Technology Brisbane, Australia.
- Vikas Agarwal, Girish Chafle, Koustuv Dasgupta, Neeran M. Karnik, Arun Kumar, Sumit Mittal, and Biplav Srivastava. 2005. Synthy: A system for end to end composition of web services. *Journal on Web Semantics: Science, Services and Agents on the World Wide Web* 3, 4 (2005), 311–339. DOI: <http://dx.doi.org/10.1016/j.websem.2005.09.002>
- Clemens Beckstein and Joachim Klausner. 1999. A Meta Level Architecture for Workflow Management. *Journal of Integrated Design and Process Science* 3, 1 (1999), 15–26.
- Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. 2003. Automatic Composition of E-services That Export Their Behavior. In *Proceedings of the First International Conference on Service-Oriented Computing (ICSOC 2003)*. Springer, 43–58. DOI: http://dx.doi.org/10.1007/978-3-540-24593-3_4
- BPMI.org and OMG. 2011. Business Process Modeling Notation - Final Specification Ver.2.0. <http://www.omg.org/spec/BPMN/2.0/PDF/>. (January 2011).
- Ronald Brachman and Hector Levesque. 2004. *Knowledge Representation and Reasoning*. Elsevier.
- Antonio Bucchiarone, Marco Pistore, Heorhi Raik, and Raman Kazhamiakin. 2011. Adaptation of service-based business processes by context-aware replanning. In *Proceedings of the 4th International Conference on Service-Oriented Computing and Applications (SOCA 2011)*. IEEE, 1–8. DOI: <http://dx.doi.org/10.1109/SOCA.2011.6166209>

¹⁴More information about the system is available at <http://www.dis.uniroma1.it/~smartpm>

- Andrea Capata, Andrea Marella, and Ruggero Russo. 2008. A Geo-based Application for the Management of Mobile Actors during Crisis Situations. In *Proceedings of the 5th International Conference on Information Systems for Crisis Response and Management (ISCRAM 2008)*.
- Fabio Casati, Stefano Ceri, Stefano Paraboschi, and Guiseppe Pozzi. 1999. Specification and implementation of exceptions in workflow management systems. *ACM Transactions on Database Systems (TODS)* 24, 3 (1999), 405–451.
- Fabio Casati and Gianpaolo Cugola. 2001. Error Handling in Process Support Systems. In *Advances in Exception Handling Techniques*. Springer-Verlag, 251–270.
- Tiziana Catarci, Massimiliano De Leoni, Andrea Marrella, Massimo Mecella, Alessandro Russo, Renate Steinmann, and Manfred Bortenschlager. 2013. WORKPAD: Process Management and Geo-Collaboration Help Disaster Response. *Using Social and Information Technologies for Disaster and Crisis Management* (2013), 33–51. DOI: <http://dx.doi.org/10.4018/978-1-4666-2788-8.ch003>
- Dickson K. W. Chiu, Qing Li, and Kamalakar Karlapalem. 2000. A Logical Framework for Exception Handling in ADOME Workflow Management System. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE '00)*. Springer-Verlag, 110–125.
- Jens Claßen, Patrick Eyerich, Gerhard Lakemeyer, and Bernhard Nebel. 2007a. Towards an Integration of Golog and Planning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*. 1846–1851.
- Jens Claßen, Yuxiao Hu, and Gerhard Lakemeyer. 2007b. A Situation-Calculus Semantics for an Expressive Fragment of PDDL. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*. 956–961.
- Giuseppe De Giacomo, Yves Lespérance, Hector Levesque, and Sebastian Sardina. 2009. In-diGolog: A High-Level Programming Language for Embedded Reasoning Agents. (2009), 31–72. DOI: http://dx.doi.org/10.1007/978-0-387-89299-3_2
- Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121, 1 (2000), 109–169.
- Giuseppe De Giacomo, Massimo Mecella, and Fabio Patrizi. 2014. Automated Service Composition Based on Behaviors: The Roman Model. In *Web Services Foundations*. Springer, 189–214. DOI: http://dx.doi.org/10.1007/978-1-4614-7518-7_8
- Giuseppe De Giacomo, Raymond Reiter, and Mikhail Soutchanski. 1998. Execution Monitoring of High-Level Robot Programs. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*. 453–465.
- Claudio Di Ciccio, Andrea Marrella, and Alessandro Russo. 2014. Knowledge-Intensive Processes: Characteristics, Requirements and Analysis of Contemporary Approaches. *Journal on Data Semantics* 4, 1 (2014), 1–29. DOI: <http://dx.doi.org/10.1007/s13740-014-0038-4>
- Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. 2013. *Fundamentals of Business Process Management* (1st ed.). Springer-Verlag Berlin Heidelberg. DOI: <http://dx.doi.org/10.1007/978-3-642-33143-5>
- Marlon Dumas, Wil M.P. van der Aalst, and Arthur H.M. ter Hofstede. 2005. *Process-Aware Information Systems: Bridging People and Software through Process Technology* (1st ed.). John Wiley & Sons. DOI: <http://dx.doi.org/10.1002/0471741442>
- Stefan Edelkamp and Jorg Hoffmann. 2004. *PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition*. Technical Report. Albert-Ludwigs-Universität Freiburg, Institut für Informatik.
- Johann Eder and Walter Liebhart. 1995. The Workflow Activity Model WAMO. In *Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIS-95)*. 87–98.
- Johann Eder and Walter Liebhart. 1996. Workflow recovery. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems (CoopIS'96)*. IEEE Computer Society, 124–134.
- Clarence A. Ellis, Simon J. Gibbs, and Gail Rein. 1991. Groupware: some issues and experiences. *Commun. ACM* 34, 1 (1991), 39–58. DOI: <http://dx.doi.org/10.1145/99977.99987>
- Hugo M. Ferreira and Diogo R. Ferreira. 2006. An Integrated Life Cycle for Workflow Management Based on Learning and Planning. *International Journal on Cooperative Information Systems* 15 (2006).
- Christian Fritz, Jorge A. Baier, and Sheila A. McIlraith. 2008. ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for Planning and Beyond. In *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*. 600–610.
- Michał Gajewski, Harald Meyer, Mariusz Momotko, Hilmar Schuschel, and Mathias Weske. 2005. Dynamic Failure Recovery of Generated Workflows. In *Proceedings of the 16th International Workshop on Database and Expert Systems Applications (DEXA 2005)*. IEEE Computer Society Press, 982–986. DOI: <http://dx.doi.org/10.1109/DEXA.2005.78>

0:40

Andrea Marrella, Massimo Mecella, Sebastian Sardina

- Hector Geffner and Blai Bonet. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers. DOI : <http://dx.doi.org/10.2200/S00513ED1V01Y201306AIM022>
- Alfonso Gerevini, Alessandro Saetti, Ivan Serina, and Paolo Toninelli. 2004. LPG-TD: a Fully Automated Planner for PDDL2.2 Domains. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*.
- Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu J. Upadhyaya. 2007. Self-healing systems - survey and synthesis. *Decision Support Systems* 42, 4 (2007), 2164–2185. DOI : <http://dx.doi.org/10.1016/j.dss.2006.06.011>
- Paul Grefen, Jochem Vonk, and Peter Apers. 2001. Global transaction support for workflow management systems: from formal specification to practical implementation. *The VLDB Journal* 10 (2001), 316–333. DOI : <http://dx.doi.org/10.1007/s007780100056>
- Claus Hagen and Gustavo Alonso. 2000. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering* 26, 10 (2000), 943–958.
- Keith Harrison-Broninski. 2013. *Human Processes: Capturing Knowledge with Processes*. BPTrends, www.bptrends.com. (November 2013).
- Sumi Helal, William Mann, Hicham El-Zabadani, Jeffrey King, Youssef Kaddoura, and Erwin Jansen. 2005. The Gator Tech Smart House: A Programmable Pervasive Space. *Computer* 38, 3 (2005). DOI : <http://dx.doi.org/10.1109/MC.2005.107>
- Malte Helmert. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26, 1 (2006), 191–246.
- Malte Helmert. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173, 5 (2009), 503–535. DOI : <http://dx.doi.org/10.1016/j.artint.2008.10.013>
- Shah Rukh Humayoun, Tiziana Catarci, Massimiliano de Leoni, Andrea Marrella, Massimo Mecella, Manfred Bortenschlager, and Renate Steinmann. 2009a. Designing Mobile Systems in Highly Dynamic Scenarios: The WORKPAD Methodology. *Knowledge, Technology & Policy* 22, 1 (2009), 25–43. DOI : <http://dx.doi.org/10.1007/s12130-009-9070-3>
- Shah Rukh Humayoun, Tiziana Catarci, Massimiliano de Leoni, Andrea Marrella, Massimo Mecella, Manfred Bortenschlager, and Renate Steinmann. 2009b. The WORKPAD User Interface and Methodology: Developing Smart and Effective Mobile Applications for Emergency Operators. In *Proceedings of the 5th International Conference on Universal Access in Human-Computer Interaction (UAHCI 2009)*. Springer Berlin Heidelberg, 343–352. DOI : http://dx.doi.org/10.1007/978-3-642-02713-0_36
- Peter Jarvis, Jonathan Moore, Jussi Stader, Ann Macintosh, Andrew Casson du Mont, and Paul Chung. 1999. Exploiting AI Technologies to Realise Adaptive Workflow Systems. In *Proceedings of the AAAI Workshop on Agent-Based Systems in the Business Context*.
- Nicholas R. Jennings, Peyman Faratin, Timothy J. Norman, P. O'Brien, Brian Odgers, and James L. Alty. 2000. Implementing a Business Process Management System using ADEPT: A Real-World Case Study. *Applied Artificial Intelligence: An International Journal* 14, 5 (2000), 421–463. DOI : <http://dx.doi.org/10.1080/088395100403379>
- Kurt Jensen and Lars Michael Kristensen. 2009. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer. DOI : <http://dx.doi.org/10.1007/b95112>
- Sandy Kemsley. 2011. The Changing Nature of Work: From Structured to Unstructured, from Controlled to Social. In *Proceedings of the 9th International Conference on Business Process Management (BPM 2011)*. DOI : http://dx.doi.org/10.1007/978-3-642-23059-2_2
- Mark Klein and Chrysanthos Dellarocas. 2000. A Knowledge-based Approach to Handling Exceptions in Workflow Systems. *Computer Supported Cooperative Work (CSCW)* 9, 3-4 (2000), 399–412. DOI : <http://dx.doi.org/10.1023/A:1008759413689>
- Andreas Lanz, Manfred Reichert, and Peter Dadam. 2011. Robust and Flexible Error Handling in the AristaFlow BPM Suite. In *Information Systems Evolution: CAiSE Forum 2010*. Springer Berlin Heidelberg, 174–189. DOI : http://dx.doi.org/10.1007/978-3-642-17722-4_13
- Edward A Lee. 2008. Cyber Physical Systems: Design Challenges. In *Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2008)*. IEEE, 363–369. DOI : <http://dx.doi.org/10.1109/ISORC.2008.25>
- Richard Lenz and Manfred Reichert. 2007. IT support for healthcare processes - premises, challenges, perspectives. *Data & Knowledge Engineering* 61, 1 (April 2007), 39–58. DOI : <http://dx.doi.org/10.1016/j.datak.2006.04.007>
- Barbara Staudt Lerner, Stefan Christov, Leon J. Osterweil, Reda Bendraou, Udo Kannengiesser, and Alexander Wise. 2010. Exception Handling Patterns for Process Modeling. *IEEE Transactions on Software Engineering* 36, 2 (2010), 162–183. DOI : <http://dx.doi.org/10.1109/TSE.2010.1>

- Frank Leymann. 2001. Managing Business Processes Via Workflow Technology. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB 2001)*. Morgan Kaufmann.
- Zongwei Luo, Amit Sheth, Krys Kochut, and John Miller. 2000. Exception Handling in Workflow Systems. *Applied Intelligence* 13, 2 (2000), 125–147. DOI: <http://dx.doi.org/10.1023/A:1008388412284>
- Andrea Marrella and Yves Lespérance. 2013a. Synthesizing a Library of Process Templates through Partial-Order Planning Algorithms. In *Proceedings of the 14th International Conference on Business Process Modeling, Development, and Support (BPMDs'13)*. 277–291. DOI: http://dx.doi.org/10.1007/978-3-642-38484-4_20
- Andrea Marrella and Yves Lespérance. 2013b. Towards a Goal-Oriented Framework for the Automatic Synthesis of Underspecified Activities in Dynamic Processes. In *Proceedings of the 6th International Conference on Service-Oriented Computing (SOCA 2013)*. 361–365. DOI: <http://dx.doi.org/10.1109/SOCA.2013.43>
- Andrea Marrella, Massimo Mecella, and Alessandro Russo. 2011. Collaboration On-the-field: Suggestions and Beyond. In *Proceedings of the 8th International Conference on Information Systems for Crisis Response and Management (ISCRAM 2011)*.
- Andrea Marrella, Massimo Mecella, and Sebastian Sardina. 2014. SmartPM: An Adaptive Process Management System through Situation Calculus, IndiGolog, and Classical Planning. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*.
- Andrea Marrella, Alessandro Russo, and Massimo Mecella. 2012. Planlets: Automatically Recovering Dynamic Processes in YAWL. In *Proceedings of the 20th International Conference on Cooperative Information Systems (CoopIS 2012)*. Springer Berlin Heidelberg, 268–286. DOI: http://dx.doi.org/10.1007/978-3-642-33606-5_17
- Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. 1998. *PDDL - The Planning Domain Definition Language*. Technical Report.
- Andreas Meyer, Sergey Smirnov, and Mathias Weske. 2011. *Data in Business Processes*. Technical Report 50. Hasso Plattner Institute at the University of Potsdam.
- Mirjam Minor, Ralph Bergmann, and Sebastian Görg. 2014. Case-based adaptation of workflows. *Information Systems* 40 (2014), 142–152. DOI: <http://dx.doi.org/10.1016/j.is.2012.11.011>
- Robert Müller, Ulrike Greiner, and Erhard Rahm. 2004. AGENT WORK: A Workflow System Supporting Rule-based Workflow Adaptation. *Data & Knowledge Engineering* 51, 2 (Nov. 2004), 223–256. DOI: <http://dx.doi.org/10.1016/j.datak.2004.03.010>
- Nicolas Mundbrod, Jens Kolb, and Manfred Reichert. 2013. Towards a System Support of Collaborative Knowledge Work. In *Proceedings of the 2012 Business Process Management Workshops*. Springer Berlin Heidelberg, 31–42. DOI: http://dx.doi.org/10.1007/978-3-642-36285-9_5
- Karen L. Myers and Pauline M. Berry. 1998. Workflow Management Systems: An AI Perspective. *AIC-SRI report* (1998).
- Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, William Murdock, Dan Wu, and Fusun Yaman. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)* 20 (2003), 379–404.
- Dana Nau, Malik Ghallab, and Paolo Traverso. 2004. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and Annapaola Marconi. 2005a. Automated Synthesis of Composite BPEL4WS Web Services. In *Proceedings of the 2005 IEEE International Conference on Web Services (ICWS'05)*. IEEE Computer Society, 293–301. DOI: <http://dx.doi.org/10.1109/ICWS.2005.27>
- Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and Annapaola Marconi. 2005b. Automated synthesis of executable web service compositions from BPEL4WS processes. In *Proceedings of the 14th International Conference on World Wide Web (WWW 2005)*. ACM, 1186–1187. DOI: <http://dx.doi.org/10.1145/1062745.1062931>
- Harald Psailer and Schahram Dustdar. 2011. A survey on self-healing systems: approaches and systems. *Computing* 91, 1 (2011), 43–73. DOI: <http://dx.doi.org/10.1007/s00607-010-0107-y>
- Frank Puhlmann and Mathias Weske. 2005. Using the π -Calculus for Formalizing Workflow Patterns. In *Proceedings of the Third International Conference on Business Process Management (BPM 2005)*. Springer Berlin Heidelberg, 153–168. DOI: http://dx.doi.org/10.1007/11538394_11
- María Dolores R-Moreno and Paul Kearney. 2002. Integrating AI Planning Techniques with Workflow Management System. *Knowledge-Based Systems* 15, 5-6 (2002).
- Ragunathan Raj Rajkumar, Insup Lee, Lui Sha, and John Stankovic. 2010. Cyber-Physical Systems: The Next Computing Revolution. In *Proceedings of the 47th Design Automation Conference (DAC 2010)*. IEEE, 731–736.

0:42

Andrea Marrella, Massimo Mecella, Sebastian Sardina

- Anand S. Rao and Michael P. Georgeff. 1995. BDI Agents: From Theory to Practice. In *Proceedings of the First International Conference on Multiagent Systems (ICMAS 95)*. 312–319.
- Manfred Reichert and Peter Dadam. 1998. ADEPTflex – Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems* 10, 2 (1998), 93–129. DOI : <http://dx.doi.org/10.1023/A:1008604709862>
- Manfred Reichert, Stefanie Rinderle, and Peter Dadam. 2003. ADEPT Workflow Management System. In *Proceedings of the 1st International Conference on Business Process Management (BPM 2003)*. Springer Berlin Heidelberg, 370–379. DOI : http://dx.doi.org/10.1007/3-540-44895-0_25
- Manfred Reichert, Stefanie Rinderle, Ulrich Kreher, and Peter Dadam. 2005. Adaptive Process Management with ADEPT2. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. 1113–1114. DOI : <http://dx.doi.org/10.1109/ICDE.2005.17>
- Manfred Reichert and Barbara Weber. 2012. *Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies*. Springer Berlin Heidelberg. DOI : <http://dx.doi.org/10.1007/978-3-642-30409-5>
- Han Reichgelt. 1991. *Knowledge Representation: An AI perspective*. Greenwood Publishing Group Inc.
- Raymond Reiter. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.
- Stefanie Rinderle, Manfred Reichert, and Peter Dadam. 2004. Correctness Criteria for Dynamic Changes in Workflow Systems: A Survey. *Data & Knowledge Engineering* 50, 1 (2004), 9–34. DOI : <http://dx.doi.org/10.1016/j.datak.2004.01.002>
- Stefanie Rinderle, Barbara Weber, Manfred Reichert, and Werner Wild. 2005. Integrating Process Learning and Process Evolution – A Semantics Based Approach. In *Proceedings of the Third International Conference on Business Process Management (BPM 2005)*. Springer Berlin Heidelberg, 252–267. DOI : http://dx.doi.org/10.1007/11538394_17
- Austin Rosenfeld. 2011. BPM: Structured vs. Unstructured. BPTrends, www.bptrends.com. (September 2011).
- Nick Russell, Wil M.P. van der Aalst, and Arthur H.M. ter Hofstede. 2006. Workflow exception patterns. In *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE 2006)*. Springer Berlin Heidelberg, 288–302. DOI : http://dx.doi.org/10.1007/11767138_20
- Shazia W. Sadiq, Wasim Sadiq, and Maria E. Orlowska. 2001. Pockets of Flexibility in Workflow Specification. In *Proceedings of the 20th International Conference on Conceptual Modeling (ER '01)*. Springer Berlin Heidelberg, 513–526. DOI : http://dx.doi.org/10.1007/3-540-45581-7_38
- Ronny Seiger, Christine Keller, Florian Niebling, and Thomas Schlegel. 2014. Modelling complex and flexible processes for smart cyber-physical environments. *Journal of Computational Science* (2014), 137–148. DOI : <http://dx.doi.org/10.1016/j.jocs.2014.07.001>
- Bart Selman, Henry A. Kautz, and Bram Cohen. 1994. Noise strategies for improving local search. In *Proceedings of the Twelfth International Conference on Artificial Intelligence (AAAI '94)*. American Association for Artificial Intelligence, 337–343.
- Arthur H.M. ter Hofstede, Wil M.P. van der Aalst, Michael Adams, and Nick Russell. 2009. *Modern Business Process Automation: YAWL and its Support Environment*. Springer. DOI : <http://dx.doi.org/10.1007/978-3-642-03121-2>
- Nick R.T.P. van Beest, Eirini Kaldeli, Pavel Bulanov, Johan C. Wortmann, and Alexander Lazovik. 2014. Automated runtime repair of business processes. *Information Systems* 39 (2014), 45–79. DOI : <http://dx.doi.org/10.1016/j.is.2013.07.003>
- Wil M.P. van Der Aalst. 1996. Three Good Reasons for Using a Petri-Net-Based Workflow Management System. In *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96)*. 179–201. DOI : http://dx.doi.org/10.1007/978-1-4615-5499-8_10
- Wil M.P. van der Aalst. 1998. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers* 8, 1 (1998), 21–66. DOI : <http://dx.doi.org/10.1142/S0218126698000043>
- Wil M.P. van der Aalst. 2013. Business Process Management: A Comprehensive Survey. *ISRN Software Engineering* (2013). DOI : <http://dx.doi.org/10.1155/2013/507984>
- Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. 2003. Workflow Patterns. *Distributed Parallel Databases* 14, 1 (2003). DOI : <http://dx.doi.org/10.1023/A:1022883727209>
- Barbara Weber, Manfred Reichert, and Stefanie Rinderle. 2008. Change Patterns and Change Support Features - Enhancing Flexibility in Process-aware Information Systems. *Data & knowledge engineering* 66, 3 (2008), 438–466. DOI : <http://dx.doi.org/10.1016/j.datak.2008.05.001>

Intelligent Process Adaptation in the SmartPM System

0:43

Barbara Weber, Werner Wild, and Ruth Breu. 2004. CBRFlow: Enabling Adaptive Workflow Management Through Conversational Case-Based Reasoning. *Proceedings of the 7th European Conference on Advances in Case-Based Reasoning (2004)*.

Mathias Weske. 2001. Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS 2001)*. IEEE. DOI : <http://dx.doi.org/10.1109/HICSS.2001.927082>

Mathias Weske. 2012. *Business Process Management: Concepts, Languages, Architectures* (2nd ed.). Springer Science & Business Media. DOI : <http://dx.doi.org/10.1007/978-3-642-28616-2>

Online Appendix to: Intelligent Process Adaptation in the SmartPM System

ANDREA MARRELLA and MASSIMO MECELLA, Sapienza Università di Roma, Italy
SEBASTIAN SARDINA, RMIT University, Australia

A process modeling language provides appropriate syntax and semantics to precisely specify business process requirements, in order to support automated process verification, validation, simulation and automation. One of the main obstacles in applying Artificial Intelligent (AI) techniques to real problems is the difficulty to model the domains. Usually, this requires that people that have developed the AI system carry out the modeling phase since the representation depends very much on a deep knowledge of the internal working of the AI tools.

To tackle the above issue, in SmartPM we implemented a GUI-based tool, called the SmartPM Definition Tool, which supports the process design activity by providing (i) a wizard-based GUI that assists the process designer in the definition of the process knowledge, and (ii) a graphical editor to design the control flow of a KiP using a subset of the BPMN 2.0 notation. The use of the SmartPM Definition Tool allows a human process designer to enter knowledge on processes without being expert of the internal working of the AI tools involved in the system.

In this appendix, we introduce the main components of the SmartPM Definition Tool, which combines a modeling language - named the SmartPM Modeling Language (SmartML) - used for modeling the contextual information in which the process is meant to run, and a graphical editor to design the control flow of a KiP using the BPMN 2.0 notation. We notice that a SmartML specification, together with the control flow of the KiP to be enacted, is translatable in situation calculus, IndiGolog and PDDL readable formats, and thus executable by the IndiGolog engine provided by SmartPM and adapted (if needed) through a state-of-the-art planning system. Finally, we show some screenshots of the SmartPM system in action while executing the KiP introduced in our running example.

A. THE SMARTPM DEFINITION TOOL

The SmartPM Definition Tool has been developed as a standard Java application, using the Java SE 7 Platform, and the JGraphX open source graphical library.¹⁵ A screenshot of the workspace of the SmartPM Definition Tool is shown in Figure 10.

The workspace is composed of a Menu Bar, a Menu Toolbar, a Modeling Canvas, a Process Elements Panel, a Context Menu and an Information View Panel. The *Menu Bar* contains several options to assist the process designer in creating, editing, configuring, analysing, maintaining and validating a process control flow built with the BPMN 2.0 syntax. The *Menu Toolbar* replaces some of the most important Menu Bar choices by presenting them in the form of selectable buttons. The *Modeling canvas* is where BPMN process elements (dragged and dropped from the *Process Elements Panel*) are placed to create and modify the control flow of a process specification. The *Information View Panel* provides a high level view of such a control flow. The *Context Menu* is a pop-up menu that appears upon a right-click mouse operation on any

¹⁵<http://www.jgraph.com/>

App-2

Andrea Marrella, Massimo Mecella, Sebastian Sardina

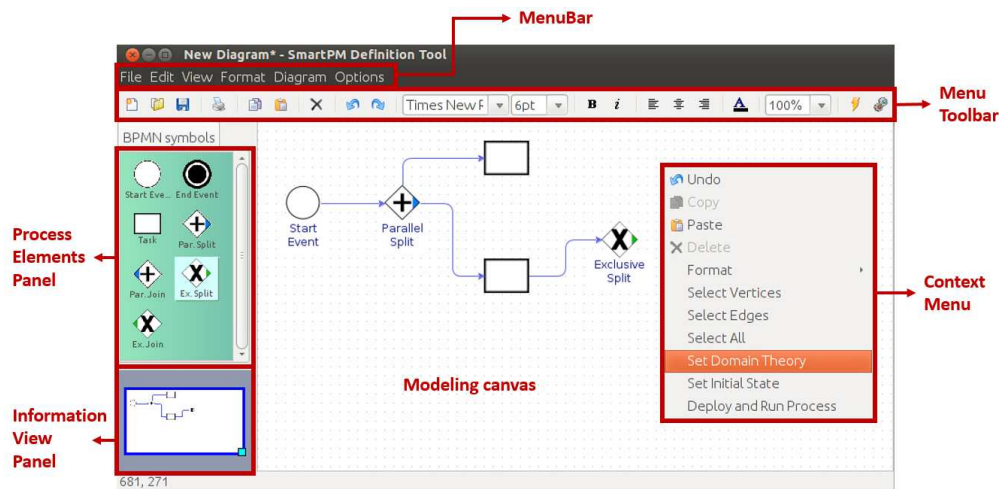


Fig. 10. The workspace provided by the SmartPM Definition Tool.

blank area of the modeling canvas. It presents several modeling options, but the most relevant are the following:

- *Set Domain Theory*: This item will display a panel (cf. Figure 11(a)) showing an aggregated perspective of data, resources, constraints and tasks involved in the current process specification. Such an information is presented as a SmartML domain theory. SmartML is a declarative language used for representing all the contextual information of the domain of concern (see the next section for its complete syntax). An *Edit* menu (cf. Figure 11(b)) provides several features for assisting the user in the definition of a valid SmartML domain theory. Specifically, it allows a process designer to define her/his own data types and process variables (in the form of *atomic terms*, see later) in SmartML, to modify the resource perspective, and to create/edit new/existing tasks that can be possibly used for the process control flow definition. Furthermore, it allows to create new exogenous events and associate their occurrence to specific contextual changes.
- *Set Initial State*: This item will display a panel where the process designer can provide an initial description of the execution context, by instantiating the SmartML domain theory with a starting condition.
- *Deploy and Run Process*: When a process is ready for being executed (i.e., a SmartML domain theory has been defined for it, an initial state has been instantiated on the domain theory and a control flow for the process has been created), the *Run Process* item allows to invoke the *XML-to-IndiGolog Parser* component (cf. Figure 5). Such a component translates the SmartML specification and the BPMN control flow of the process in a IndiGolog program and passes it to the *Execution Layer* for its deployment and execution.

In the following two sections, we show the main ingredients required to create a new process specification with the SmartPM Definition Tool. Specifically, we describe the basics of the SmartML notation and the relevant subset of BPMN 2.0 used to build control flows of processes.

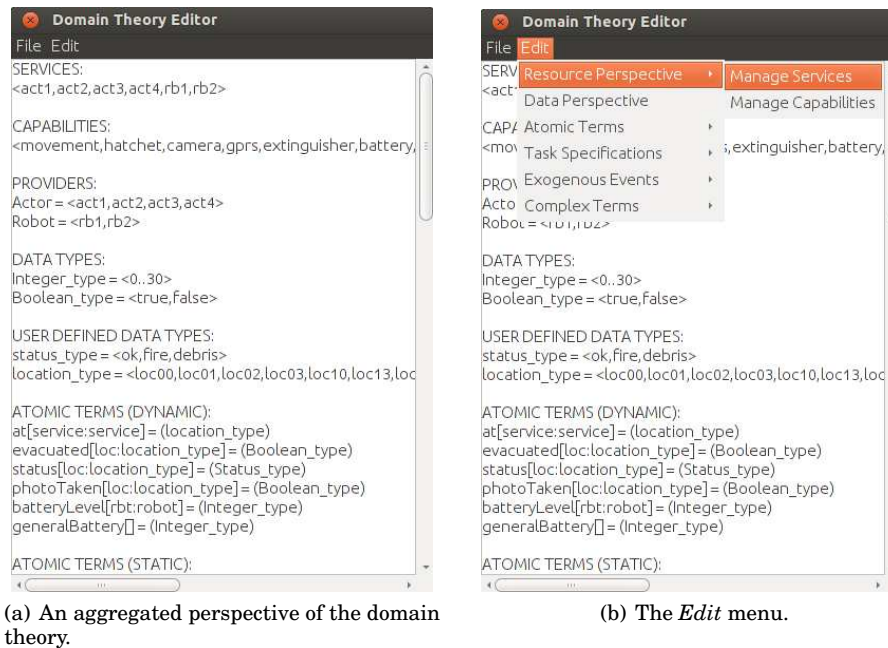


Fig. 11. Overview of the Domain Theory editor.

A.1. SmartML: The SmartPM Modeling Language

The synthesis of a KiP in SmartPM requires a tight integration of process activities and contextual data in which the process is embedded in. The context is represented in the form of a *Domain Theory* D , that involves capturing a set of tasks $t_i \in T$ (with $i \in 1..n$) and supporting information, such as the people/agents that may be involved in performing the process (roles or participants), the data and so forth. In SmartPM we adopt a service-based approach to process management. Thus, tasks are executed by *services*, such as software applications, human actors, robots, etc.

Tasks are collected in a specific repository, and each task can be considered as a single step that consumes input data and produces output data. Data are represented through some ground *atomic terms* $v_1[y_1], v_2[y_2], \dots, v_m[y_m] \in V$ that range over a set of tuples (i.e., unordered sets of zero or more attributes) y_1, y_2, \dots, y_m of *data objects*, defined over some *data types*. In short, a data object depicts an entity of interest.

Some data types are pre-specified and used for representing the *resource perspective* of the process. For example, in our scenario we need to define data objects for representing services (e.g., data type $Service = \{act1, act2, act3, act4, rb1, rb2\}$) and capabilities (e.g., data type $Capability = \{extinguisher, movement, \dots, hatchet\}$).

Resource Perspective :

```

Service = {act1,act2,act3,act4,rb1,rb2}
Capability = {movement,hatchet,camera,gprs,extinguisher,battery,digger,
powerpack}

```

Since the data types *Service* and *Capability* are pre-defined for being used in the framework, a process designer is just required to provide values (i.e., to associate data objects) to the above types. Furthermore, the data type *Service* can be customized de-

App-4

Andrea Marrella, Massimo Mecella, Sebastian Sardina

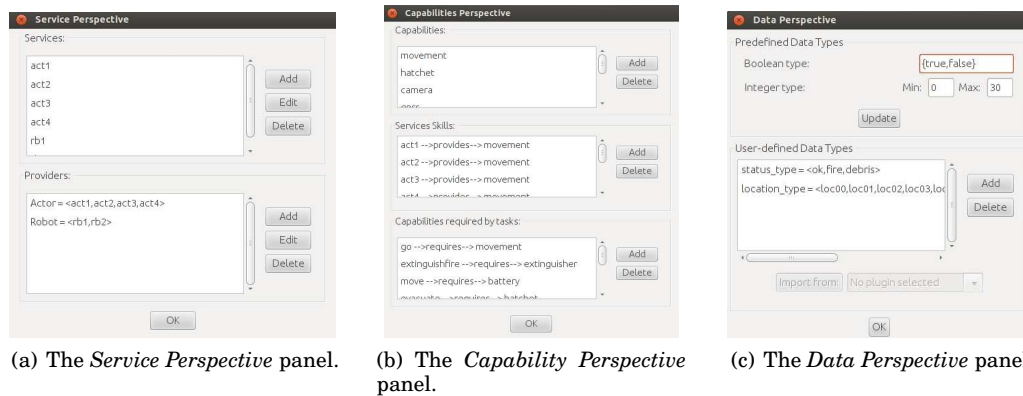


Fig. 12. Panels for customizing the resource and data perspectives.

pending on the specific service provider. Therefore, since our contextual scenario involves both human actors and robots, a process designer can specialize the data type *Service* by introducing two further data types *Actor* and *Robot*. Their data objects must be a subset of the ones introduced through the data type *Service*.

Service Providers :

Actor = {act1, act2, act3, act4}
 Robot = {rb1, rb2}

Figures 12(a) and 12(b) show a couple of screenshots of the *Service Perspective* panel and of the *Capability Perspective* panel, which are used for updating the resources (i.e., services, providers and capabilities) involved in the process.

In order to describe the contextual scenario in which the process will be enacted, some *domain-dependent* data types need to be defined. In our example, we need of a data type *Location.type* = {loc00, loc10, ..., loc33, lost} for representing locations in the area (the special constant 'lost' is used to indicate that we lose track of an actor location) and of a data type *Status.type* = {ok, fire, debris}, which enumerates all the possible states of a location (e.g, a location can be out of danger, on fire or buried by debris).

Domain-dependent Data Types :

Location_type = {loc00, loc10, loc20, loc30, loc01, loc11, loc02, loc03, loc13, loc23, loc31, loc32, loc33, lost}
 Status_type = {ok, fire, debris}

Under this representation, we consider possible values of a data type as constant symbols that univocally identify data objects in the scenario of interest. Each tuple y_j may contain one or more data objects belonging to different data types. The domain $dom(v_j[y_j])$ over which a term is interpreted can be of various types:

- **Boolean_type:** $dom(v_j[y_j]) = \{true, false\}$;
- **Integer_type:** $dom(v_j[y_j]) = \{x...y\}$ s.t. $x, y \in \mathbb{Z} \wedge (x \leq y)$;
- **Functional:** the domain contains a fixed number of data objects of a designated type.

The data type *Boolean_type* (and its respective data objects *true* and *false*) and the data type *Integer_type* are pre-defined in the framework. However, since integer num-

bers form a countably infinite set, there is the need to set a lower and an upper bound to specify which finite subset of integers is relevant for the case to deal with. In the below example, we are considering the subset of the integers from 0 to 30.

Pre-Defined Data Types :

```
Boolean_type = {true,false}
Integer_type = {0,1,2,3,4,...,30}
```

The data perspective can be easily customized through the *Data Perspective* panel (cf. Figure 12(c)) provided by the SmartPM Definition Tool.

Atomic terms can be used to express properties of domain objects (and relations over objects), and argument types of a term - taken from the set of data types previously defined¹⁶ - represent the finite domains over which the atomic term is interpreted.

Some atomic terms are pre-specified for being used in the framework. For example, since each task has to be assigned to a service that provides all of the skills required for executing that task, there is the need to consider the services “capabilities”. This can be done through a boolean term *provides[*srv:Service,cap:Capability*]* that is true if the capability *cap* is provided by *srv* and false otherwise. At the same way, a boolean term *requires[*task:Task,cap:Capability*]* is needed for specifying which capabilities are required for executing a specific task.¹⁷ The boolean terms *provides* and *requires* can be customized through the *Capability Perspective* panel (cf. Figure 12(b)).

Pre-Defined Terms :

```
provides[srv:Service,cap:Capability] = (bool:Boolean_type)
requires[task:Task,cap:Capability] = (bool:Boolean_type)
```

Moreover, we need boolean terms for indicating if people have been evacuated from a location (e.g., *evacuated[*loc:Location_type*] = (bool:Boolean_type)*) or if some picture has been collected in a specific location (e.g., *photoTaken[*loc:Location_type*] = (bool:Boolean_type)*), integer terms for representing the battery charge level of each robot (e.g., *batteryLevel[*rb:Robot*] = (int:Integer_type)*) and functional terms for recording the position of each actor and robot (e.g., *at[*srv:Service*] = (loc:Location_type)*) in the area or for representing the current status of each location (e.g., *status[*loc:Location_type*] = (st:Status_type)*).

Note that some terms may be used as *constant values*. For example, the atomic term *generalBattery[] = (int:Integer_type)* reflects the battery charge level stored in the power pack and used for recharging the battery of each robot. The term *batteryRecharging[] = (int:Integer_type)* indicates the amount of battery that is charged after each recharging action. The terms *moveStep[*loc1:Location_type,loc2:Location_type*] = (int:Integer_type)* and *debrisStep[] = (int:Integer_type)* indicate the amount of battery consumed respectively after the robot has been moved from a location *loc1* to a location *loc2*, and after having removed debris from a specific location.

Finally, atomic terms can be also used to express static relations over objects. The term *neigh[*loc1:Location_type,loc2:Location_type*] = (bool:Boolean_type)* indicates all adjacent locations in the area (for example, *neigh[*loc00,loc01*] = true*), while the term *covered[*loc:Location_type*] = (bool:Boolean_type)* reflects the locations covered by the main base network.

¹⁶Predefined data types, like *Boolean_type* and *Integer_type*, can not be used as arguments of an atomic term.

¹⁷The special data type *Task* will be defined later and can be used only as argument of the pre-defined term *requires*.

App-6

Andrea Marrella, Massimo Mecella, Sebastian Sardina

The process designer can decide which atomic terms s/he consider **relevant** for being monitored during the process enactment, in order to check if their actual value becomes misaligned with the desired one.

Atomic Terms :

Relevant for Adaptation :

```
at[act:Actor] = (loc:Location_type)
evacuated[loc:Location_type] = (bool:Boolean_type)
status[loc:Location_type] = (st:Status_type)
```

Not Relevant for Adaptation :

```
at[rb:Robot] = (loc:Location_type)
batteryLevel[rb:Robot] = (int:Integer_type)
photoTaken[loc:Location_type] = (bool:Boolean_type)
generalBattery[] = (int:Integer_type)
batteryRecharging[] = (int:Integer_type)
moveStep[loc1:Location_type,loc2:Location_type] = (int:Integer_type)
debrisStep[] = (int:Integer_type)
neigh[loc1:Location_type,loc2:Location_type] = (bool:Boolean_type)
covered[loc:Location_type] = (bool:Boolean_type)
```

The screenshots in Figures 13(a) and 13(b) show, respectively, the list of atomic terms currently defined in the domain theory and a panel that helps the process designer in creating/updating a new/existing atomic term.

In addition to atomic terms, the designer can also define *complex terms*. They are declared as basic atomic terms, with the additional specification of a well-formed first-order formula that determines the truth value for the complex term. In our case study, we need to express that an actor is connected to the network if s/he is in a covered location or if s/he is in a location adjacent to a location where a robot is located:

```
isConnected(act:Actor) {
  EXISTS(l1:Location_type, l2:Location_type, rbt:Robot).
    (at(act)==l1) AND (Covered(l1)==true OR
      (at(rbt)==l2 AND (Neigh(l1,l2)==true OR (l1==l2))))
}
```

The interpretation of complex terms derives from the corresponding first-order formula and is enacted at run-time by the IndiGolog interpreter.

A formula describing a complex term can be negated (NOT) and existentially or universally quantified (EXISTS and FORALL). Note that a complex term can be used in the range of a task precondition, but it can not appear in task effects and can not involve recursion.

In order to build a well-formed first-order formula, the SmartPM Definition Tool provides an editor (cf. Figure 13(c)) that integrates a *syntax checker* to verify on the fly if the formula under construction is compliant with the first-order logic syntax.

Concerning the definition of process tasks, the process designer is required to specify which tasks are applicable to the dynamic scenario under study. Such tasks will be stored in a specific *tasks repository*, and can be used for designing the control flow of the process and for adaptation purposes.

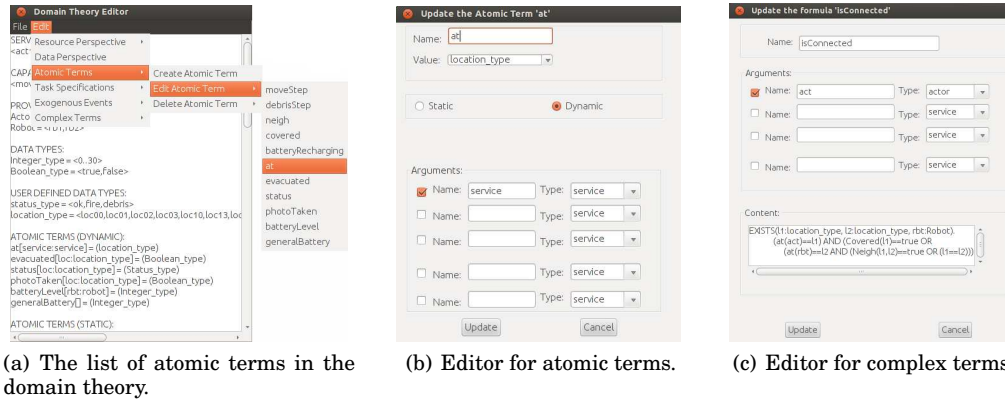


Fig. 13. Panels for customizing atomic terms and complex formulas.

Tasks Repository :

Tasks = {*go*, *move*, *takephoto*, *evacuate*, *updatestatus*, *extinguishfire*, *chargebattery*, *removeDebris*, *synchronize*}

Each task is annotated with *preconditions* and *effects*. Preconditions are logical constraints defined as a conjunction of atomic terms, and they can be used to constrain the task assignment and must be satisfied before the task is applied, while effects establish the outcome of a task after its execution.

Definition A.1. A task $t[x] \in T$ is a tuple $t = (Act_t, x, Pre_t, Eff_t)$ that consists of:

- the name Act_t of the action involved in the enactment of the task (it often coincides with the task itself);
- a tuple of data objects x as input parameters;
- a set of preconditions Pre_t , represented as the conjunction of k atomic conditions defined over some specific terms, $Pre_t = \bigwedge_{l \in 1..k} pre_{t_l}$. Each pre_{t_l} can be represented as $\{v_j[y_j] \text{ op expr}\}$, where:
 - $v_j[y_j] \in V$ is an atomic term, with $y_j \subseteq x$, i.e., admissible data objects for y_j need to be defined as task input parameters;
 - An **expr** can be a boolean value (if v_j is a boolean term); an input parameter identified by a data object (if v_j is a functional term); an integer number or an expression involving integer numbers and/or terms, combined with the arithmetic operators $\{+, -\}$ (if v_j is a integer term);
 - The condition **op** can be expressed as the equality ($==$) between boolean terms or functional terms and an admissible **expr**. On the contrary, if v_j is a integer term, it is possible to define the **op** condition as an expression that make use of relational binary comparison operators ($<$, $>$, $=$, \leq , \geq) and involve integer numbers and/or integer terms in the **expr** field.
- a set of deterministic effects Eff_t , represented as the conjunction of h atomic conditions defined over some specific terms, $Eff_t = \bigwedge_{l \in 1..h} eff_{t_l}$. Each eff_{t_l} (with $l \in 1..h$) can be represented as $\{v_j[y_j] \text{ op expr}\}$, where:
 - $v_j[y_j] \in V$ and **expr** are defined as for preconditions.
 - The condition **op** may include assignment expressions to update the values of integer terms. A numeric effect consists of an assignment operator, the integer term to be updated and a integer number or a numeric expression. Assignment

operators include (i) direct assignment (=), to assign to a integer term a value defined by an integer number; (ii) relative assignments, which can be used to increase (+=) or decrease (-=) the value of a integer term (additive assignment effects).

Note that if no preconditions are specified, then the task is always executable. Moreover, the process designer is required to make explicit if a task effect can be considered as *supposed* or *automatic*. A *supposed effect* indicates that the service executing the task must return a physical outcome for the supposed effect at run-time, that can be or not can be equal to the one declared during the task definition. If a supposed effect involves a relevant term, it is clear that the outcome returned assumes a great value for monitoring purposes. Otherwise, if an effect is flagged as *automatic*, when the task completes, its effect is automatically applied without the need to consider any physical outcome.

For example, the task GO involves two input parameters *from* and *to* of type *Location.type*, representing a starting and an arrival location. An instance of this task can be executed only if the actor that will execute it at run-time is at the starting location *from* and provides the required capabilities for executing the task GO. Consider that in SmartML we make use of a constant symbol *SRVC* to identify the service that will execute a specific task at run-time. As a consequence of task execution, the actor moves from the starting to the arrival location, and this is reflected by assigning to the functional term *at[SRVC]* the value *to* in the effect. The fact that the effect of the task GO is *supposed* means that after task execution, the service must return the real outcome indicating her/his final position.

Each task $t_i \in T$ together with its preconditions, effects and parameters can be represented as a XML annotation:

Description of the task go :

```
<task>
  <name>go</name>
  <parameters>
    <arg>from - Location_type</arg>
    <arg>to - Location_type</arg>
  </parameters>
  <precondition>at[SRVC] == from AND isConnected[PRT] == true</precondition>
  <effects>
    <supposed>at[SRVC] = to</supposed>
  </effects>
</task>
```

Let us now analyze the task MOVE. Similarly to the task GO, it involves two input parameters *from* and *to* of type *Location.type*, that represent the starting and arrival locations given in input to a robot.

Description of the task move :

```
<task>
  <name>move</name>
  <parameters>
    <arg>from - Location_type</arg>
    <arg>to - Location_type</arg>
  </parameters>
  <precondition>at[SRVC] == from AND batteryLevel[SRVC] >= moveStep[from,to]
```

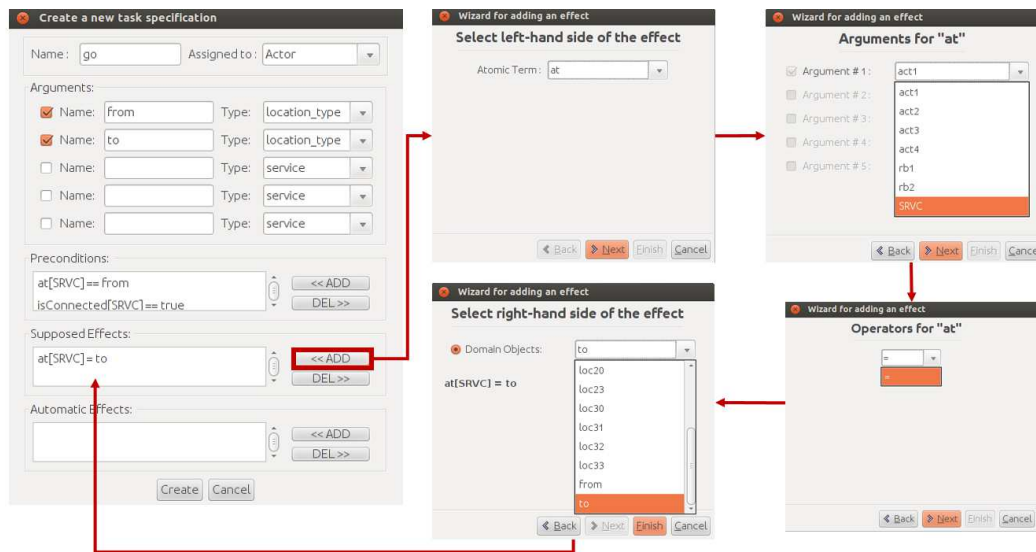


Fig. 14. The wizard-based editor to build tasks specifications.

```

</precondition>
<effects>
  <supposed>at [SRVC] = to</supposed>
  <automatic>batteryLevel [SRVC] -= moveStep[from,to]</automatic>
</effects>
</task>

```

An instance of the task MOVE can be executed only if the robot *SRVC* is at the starting location *from* and provides enough battery charge for executing the movement. As a consequence of task execution, the robot moves from the starting to the arrival location, and this is reflected by assigning to the functional term *at[SRVC]* the value *to* in the effect. This first effect of MOVE has been flagged as *supposed*, meaning that the robot *SRVC* that will execute the MOVE task at run-time must return the real outcome indicating its final position. However, since *at[SRVC]* is not considered as a relevant term (when *SRVC* is a robot, see above), if the final position of the robot will differ with the one declared at design-time, no adaptation mechanism will be triggered. The second effect of MOVE is flagged as *automatic*, and states that after the execution of the MOVE task the battery level of the robot *SRVC* will be automatically decreased of a fixed quantity, corresponding to *moveStep[from,to]*.

As shown in Figure 14, the SmartPM Definition Tool provides a wizard-based editor to build a task specification and to define the single conditions composing the task preconditions and effects.

We can also represent exogenous events with SmartML. They are events coming from the external environment that modify asynchronously atomic terms at run-time. In our case study, we can deal with four different exogenous events:

Exogenous Events :

```
Ex_events = {photoLost, fireRisk, rockSlide, push}
```

The definition of an exogenous event is similar to a task definition in SmartML. However, to define an exogenous event there is no need to specify any precondition, while

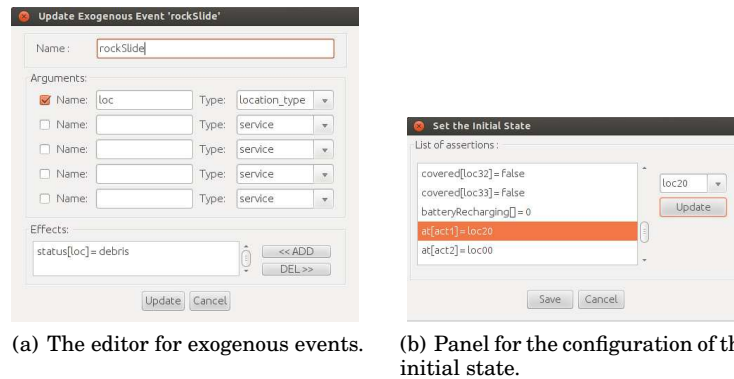


Fig. 15. Panels for customizing exogenous events and for configuring the initial state.

effects can only be considered as *automatic* (i.e., they are automatically applied to the involved terms when the exogenous event is caught). For example, the exogenous event $ROCKSLIDE(loc)$ alerts about a rock slide collapsed in location loc , and its effect modifies the value of the atomic term $Status[loc]$ to the value 'debris'.

Description of the exogenous event rockslide :

```
<ex-event>
  <name>rockSlide</name>
  <parameters>
    <arg>loc - Location_type</arg>
  </parameters>
  <effects>
    <automatic>status[loc] = debris</automatic>
  </effects>
</ex-event>
```

As for the tasks definition, the SmartPM Definition Tool provides a wizard-based editor to build exogenous event specifications (cf. Figure 15(a)).

Once a process is ready for being executed (i.e., the process designer has completed the definition of the domain theory and of the process control flow - cf. the following section), the last step before executing the process consists of instantiating the domain theory D with a *starting state*, which reflects different assignment of values to the atomic terms. By selecting the item *Set Initial State* from the *Edit* menu (cf. Figure 10), a new panel where the process designer can provide an initial description of the execution context is opened (cf. Figure 15(b)). We assume complete information about the starting state. Basically, this means we force the process designer to instantiate every atomic term with an admissible value that represent what is known in the starting state about the dynamic scenario. Specifically, the starting state is a conjunction $\{v_1[y_1] == val_1 \wedge v_2[y_2] = val_2 \dots \wedge v_j[y_j] == val_j\}$, where val_j (with $j \in 1..m$) represents the j -th value assigned to the j -th atomic term.

A.2. On defining the control flow of a KiP

Starting from a domain theory D and a set of tasks T , the control flow of a KiP in SmartPM can be defined through the Business Process Modeling Notation (BPMN). The notation has been released to the public in May 2004 by the BPMI Notation

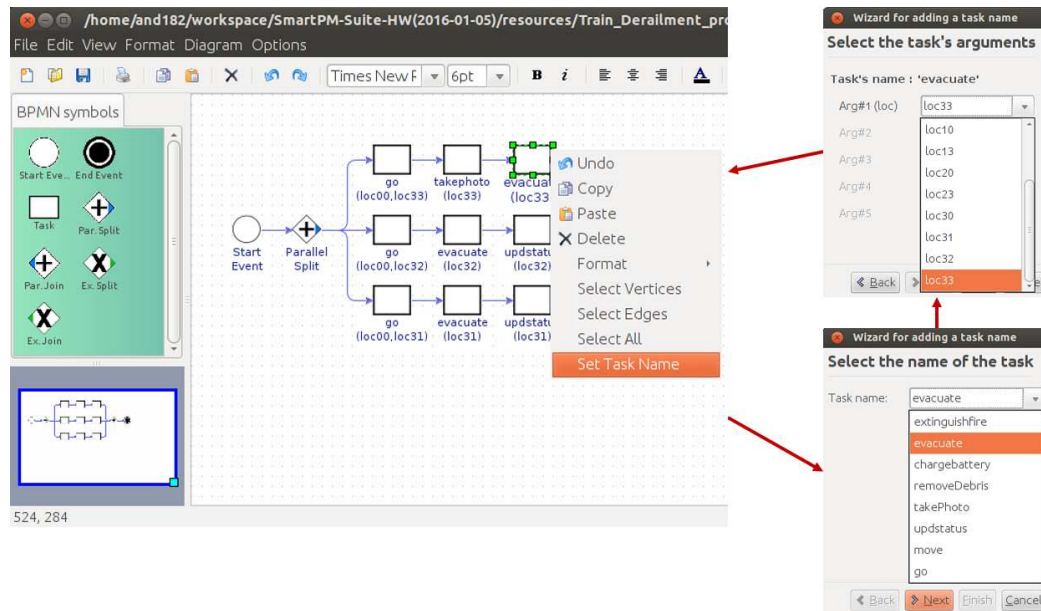


Fig. 16. Associating task specifications to a KiP.

Working Group and was adopted as OMG standard¹⁸ for business process modeling in February 2006. BPMN provides a graphical notation for specifying business processes based on a flowcharting technique similar to activity diagrams from UML. In January 2001, the version 2.0 of the language has been released. If compared to the previous specifications of the language, which provided only verbal descriptions of the graphic notations elements and modeling rules, BPMN 2.0 received a formal definition in the form of a meta-model, that defines the abstract syntax and semantics of the modeling constructs.

From a formal point of view, in SmartPM we define a KiP as a directed graph consisting of tasks, gateways, events and transitions between them.

Definition A.2. Given a domain theory D and a set of tasks T , a KiP P is a tuple (N,L) where:

- $N = T \cup E \cup W \cup X$ is a finite set of nodes, such that :
 - T is a set of task instances, i.e., occurrences of a specific task $t \in T$ in the range of the KiP;
 - E is a finite set of events, that consists of a single start event \circ and a single end event \odot ;
 - $W = W_{PS} \cup W_{PJ}$ is a finite set of parallel gateways, represented in the control flow with the \diamond shape with a “+” marker inside.
 - $X = X_{ES} \cup X_{EJ}$ is a finite set of exclusive gateways, represented in the control flow with the \diamond shape with a “X” marker inside.
- $L = L_T \cup L_E \cup L_{W_{PS}} \cup L_{W_{PJ}} \cup L_{X_{ES}} \cup L_{X_{EJ}}$ is a finite set of transitions connecting events, task instances and gateways:
 - $L_T : T \rightarrow (T \cup W_{PS} \cup W_{PJ} \cup X_{ES} \cup X_{EJ} \cup \odot)$
 - $L_E : \circ \rightarrow (T \cup W_{PS} \cup X_{ES} \cup \odot)$

¹⁸<http://www.omg.org/spec/BPMN/2.0/>

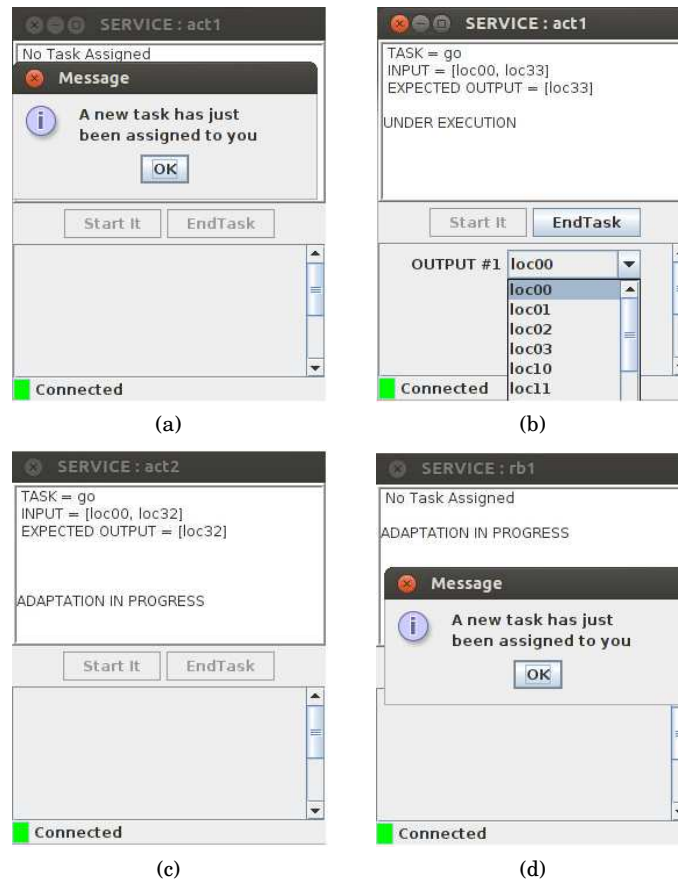


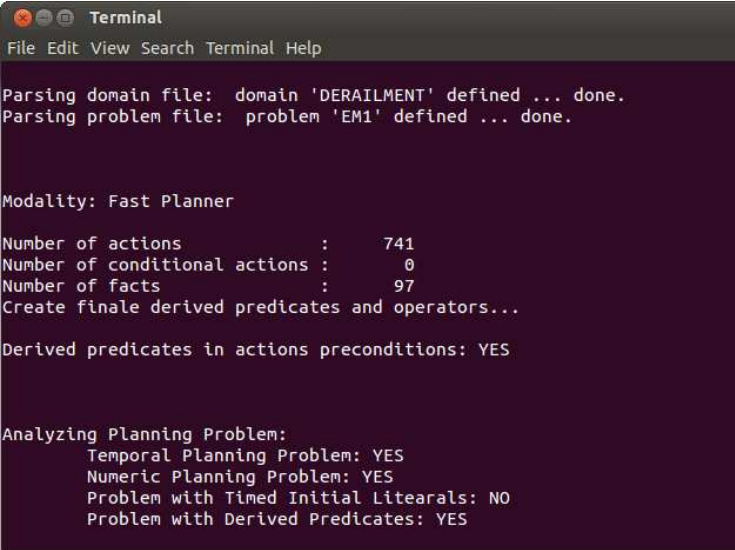
Fig. 18. The Task Handler of SmartPM.

Figure 17 shows the main window of the IndiGolog engine showing the log of all actions exchanged between the engine and involved services (i.e., virtualizations of process services). In the screenshot, we can identify an ASSIGN action that assigns to service *act2* the task $GO(id_4, [loc00, loc32], [loc32])$.

In Figure 18, we provide some screenshots of the Task Handler module used by process services to execute process tasks. Specifically, when the IndiGolog PMS assigns a task to a process service, this event is notified to the task handler of the selected service through a popup window (cf. Figure 18(a)).

When a process service is ready to start a task, it pushes the button *Start It*, and a READYTOSTART action is sent back to the IndiGolog engine. If we analyze the code in Figure 17, we can note the presence of a row =====> EXOGENOUS EVENT, representing the fact that the PMS has captured the READYTOSTART action sent by the process service's Task Handler. In response, the IndiGolog engine will instruct the process service to start the task execution through a START action (cf. the bottom part of Figure 17).

Now, let us suppose that *act1* has been instructed to start the task $GO(id_1, [loc00, loc33], [loc33])$. *act1* can select one of the valid outcomes (i.e., a list of *Location.type* data objects) and then push the button *End Task* when the task is com-



```

Terminal
File Edit View Search Terminal Help

Parsing domain file: domain 'DERAILMENT' defined ... done.
Parsing problem file: problem 'EM1' defined ... done.

Modality: Fast Planner

Number of actions      :      741
Number of conditional actions :      0
Number of facts        :      97
Create finale derived predicates and operators...

Derived predicates in actions preconditions: YES

Analyzing Planning Problem:
Temporal Planning Problem: YES
Numeric Planning Problem: YES
Problem with Timed Initial Literals: NO
Problem with Derived Predicates: YES

```

Fig. 19. The main window of the LPG-td planner.

pleted (cf. Figure 18(b)). If the outcome provided by *act1* is different from the one expected (meaning that *act1* has reached a different location than the one desired), the IndiGolog engine senses the deviation, builds a planning problem that reflects the gap between physical and expected reality and launches the LPG-TD planner (cf. Figure 19), which is in charge to synthesize the recovery plan. Note that the deviation we are analyzing is the same shown in our case study; i.e., we are supposing that *act1* has reached location *loc03* rather than location *loc33*.

During the synthesis and execution of the recovery plan, every running task is interrupted (cf. Figure 18(c)). When the planner finds a recovery procedure, it is passed back to the Synchronization component, that converts it in a executable IndiGolog process and sends it to the IndiGolog PMS for its enactment.

Since all the actors/robots need to be continually inter-connected to execute the process, the planner finds a recovery procedure that first instructs the robots to move in specific positions for maintaining the network connection, and then re-assigns the task $GO(loc03, loc33)$ to *act1*. In Figure 18(d) it is shown the assignment of a recovery task to the robot service *rb1*. Specifically, *rb1* executes the first task of the recovery procedure dealing with the deviation.

We conclude this appendix by pointing out that while other approaches and systems rely on pre-defines rules to specify the exact behaviors when special events are triggered, here we simply model (a subset of) the running environment and the actions' effects, without considering any possible exceptional event. We argue that, in most of cases, modeling the environment, even in detail, is easier than modeling all possible exceptions.